# SYSTEM, METHOD AND SOFTWARE FOR CREATING OR MAINTAINING DISTRIBUTED TRANSPARENT PERSISTENCE OF COMPLEX DATA OBJECTS AND THEIR DATA RELATIONSHIPS

This is a continuation of U.S. Provisional Patent No. 60/308,065; filed 7/26/01; and U.S. Provisional Patent No. 60/312,536; filed 8/15/01; and, U.S. Provisional Patent No. 60/316,075, filed 08/30/01.

## FIELD OF THE INVENTION

The field of the present invention relates generally to computer systems, computer data stores and to methods and software for accessing and utilizing data stores. More particularly, the present invention relates to a system, methods and software for creating or maintaining distributed transparent persistence of complex data objects and associated data stores. In one aspect, the invention also relates to an application programming object capable of creating or maintaining distributed transparent persistence of data objects or data object graphs without the necessity of inserting any byte codes or modification of the object graph. Virtually any java object or object praph can be transparently persisted. Further, copies of a data graph or of a portion of the data graph can be automatically reconciled and changes persisted without any persistence coding in the object model.

## BACKGROUND OF THE INVENTION

Systems for accessing data stores from object oriented languages have been used for many years. A frequent approach to accomplish access of data stores involves writing and embedding custom access code within an object application needing the access. This approach is generally limited to having the custom code access only a single relational table within a relational database or similar construct within any other data store (hereinafter collectively “data store"). Under the circumstances where a developer has control over the design and creation of a data store from its inception, it is possible to design and store meaningful information in a single table. Such design opportunities are usually rare, however.

Generally, the methods for producing persistence for a data object, complex data object or a data store conflict with the goals of producing pure object application models where the object models do not include persistence objects or persistence byte code. Particular difficulties exist in a distributed environment since an object application model may exist in one or more of a computer's memory, an application data store and an application information storage repository that may be independent of the data store organization or object definitions. Advancements in the art have been made with respect to tools for conveniently mapping objects to systems of tables and maps in order to expedite accessing, changing and updating data stores. See, for example, U.S. Patent 5,857,197 (and its associated programming interfaces ("APIs")) describes tools for translating object data to relational data, relational data to object data, and object data to object data to expedite the use of data stores. The BMP and the CMP Installer portions of CocoAdmin tool in the CocoBase™ Enterprise for O/R Binary Software (Thought, Inc. 657 Mission Street Suite 202, San Francisco, CA 94105 http://www.thoughtinc.com,) provide means for providing persistence in the EJB environment.

Persistence problems arise with the creation, access, changing or deleting of an object application model that utilizes such data stores. The object application model may be distributed over multiple physical computer machine locations or even distributed over multiple Internet website locations that may be independent of the data stores. The object application model may utilize a different set of data objects or different set of definitions for relationships between data objects than that of one or more of its data sources. In most situations, the respective structures of the data sources and of the object applications model simply do not conveniently allow for mapping, accessing or changing of an overall schema of application data objects as well as any associated definitions of relationships between two or more data objects or elements within a data object.

Importantly, relationships may exist between a data object and one or more of the other data objects found in the object application model or in a data object of the data source. A relationship between one data object and another data object or with a data source may be member selected from the group of three relationship types consisting of 1 to 1 (1-1), 1 to many (1-M) or many to many (M-M). Complex combinations of these relationships may exist as a data object relationships definition for a given data object. These relationships are described or illustrated in further detail later in this document.

Objects may logically span multiple relational tables or multiple object databases, and may even be distributed over a logical (or hypothetical) computer system involving multiple physically independent computer systems or even multiple website locations. Creating, accessing, maintaining or updating an object application model can require working with multiple translation modules and require tedious and repetitive updating of multiple individual computer systems or multiple data sources in order to do useful work and keep the object application model synchronized. Such approaches are both costly and unwieldy in terms of computing and development resources, particularly with respect to Internet based electronic commerce (eCommerce) object application models.

Data objects of an object application model are often a feature of eCommerce object programming applications, where information is obtained from a data source and the data is defined as a data object (e.g., as a Java class) for use with another computer application. In practice, a data object or model of data objects may exist only in the random access memory of a computer memory system, or may be saved to either a data source or to some other type of retrievable information repository. A programmer or administrator of an object data application cannot easily access or track the overall model or diagram of data objects for an object application model or some of its specific elements. Unfortunately, tools for accessing and persisting data objects and associated data object relationships of a complex data object graph model have not been well implemented in the field of object language programming.

A computer application can execute one or more of the following non-limiting actions with respect to one or more of the members selected from the group consisting of data, a data object, and a data object definition: access data, change data, create data, create a new relationship between one or more data objects by creating or changing at least one data object relationship definition, change or delete a relationship between one or more data objects by changing or deleting at least one data object relationship definition, access a data object relationship definition and use its parameters to access a data source or a data object, and access one or more data object relationship definitions or data objects to create a new data object or data object relationship. Any changes executed by a computer application with respect to one or more of the members selected from the group consisting of data, data object or data object definition may need to be properly persisted (permanently stored) to preserve any changes to one or more of the members selected from the group consisting of data, a data object and a data object definition.

A data object and an associated data object relationship definition may be represented by a complex data object graph ("CDOG"). A CDOG, for the purposes of this document, may be thought of as a computer program data object graph that represents a data object having at least one relationship with at least one other data object or with itself via a circular link. When the data object of a CDOG is implemented in the Java computer program language, the CDOG may be further defined as being a Java Data Object Graph ("JDOG").

There are needs for software, methods and systems that can easily detect and persist any changes to at least one member selected from the group consisting of a data object, any data associated with the related object, or any associated CDOG definition (i.e., an changes to the data object, data or to a relationship of the data object with another data object). For example, there is a need to be able access a pure object model definition from a repository based O/R mapping tool file or from a modeling tool repository file and provide persistence for the object model without inserting any byte code or additional objects into the object model.

Accordingly, there is a strong need in the art for a computer applications programmer tool designed to assist a programmer or administrator in the actions of providing persistence for data objects or data object graphs when deleting, inactivating or updating a CDOG, wherein the computer applications programmer tool can be configured to automatically reconcile all or a portion of a CDOG and copies thereof on a distributed environment when data objects or relationships are deleted, inactivated or updated for a CDOG. A particularly strong need exists for such a tool having the further ability to be configured to persist, propagate and reflect system wide (in a local or distributed computer system) any such changes to a CDOG instance to all instances of the CDOG and to all instances of associated data, data objects and data object relationships.

**DEFINITIONS**

The following non-exhaustive list of definitions is used herein to define terms that may otherwise be confusing or can sometimes have multiple meanings. Each occurrence of a defined term in the above text, in the text that follows, or in the claims of this document, is to be given the meaning ascribed to it in the list of definitions below.

"Instance" as referred to in this document in the context of computer software applications is a single occurrence of a software logical element in the memory of a computer system, such as a "class", an "object", a "data object", and the like.

"Class" as referred to in this document in the context of computer software applications is a logic unit in a computer application or a computer software program where the application or program is based upon an objected oriented programming language (*e.g.*, Java). In practice, a class is a logical unit used as a logical template in an object oriented language from which to allocate new instances of objects.

"Object" as used in the context of this document is a general term referring to a logic unit in a computer application or a computer software program where the application or program is based upon an objected oriented programming language (*e.g.*, Java). The term "object" may ordinarily be used interchangeably with the term "class" as a template or as an instance depending on the context.

"Data object" as referred to in the context of this document represents the concept of the occurrence of an object that holds data within a specific computer application domain and is likely to have its contents stored in a persistent data source of a computer system (e.g., a database server, a binary file, a text file, or even in a combination of two or more of such a persistent data sources of a computer system). A data object may exist as an independent data object without any relationship to any other data object or it may have one or more relationships with itself or with one or more other data objects.

"Complex data object" (or "CDO") as used in the context of this document refers to the occurrence of a data object that has at least one or more relationships with itself, or at least one or more relationships with one or more other data object(s). In a given instance of a CDO at least one relationship is populated as a link, as defined below. A CDO may have a multiplicity of different relationships with itself or with one or more additional CDOs.

"Relationship" or "data relationship" as used in the context of a CDO refers to the type of logical combination that occurs between a data object with itself, or refers to the type of logical combination that occurs between a data object and at least one another data object. Among other references or descriptions, such a relationship is always referred to or partially described by a "relationship type". This term is used in an object

oriented language context to reference or describe any expectations, actions and limitations possible between two or more data objects.

"Relationship type" in the context of this document is a label that specifies the possible multiple combinations that can occur between a CDO and itself or with at least one other CDO. The possible relationship type labels are 1-1 (one to one), 1-M (one to many) and M-M (many to many). A given CDO may be simultaneously related to more than one other CDO through several different types of relationship.

"Link" as used in this document with respect to a CDO identifies a particular occurrence of a relationship between a CDO and itself, between a CDO and another CDO. The occurrence of at least one populated link results in an instance of the CDO.

"Circular link" as used in this document with respect to a CDO identifies a particular occurrence of a relationship between a CDO and itself that may be direct or indirect (*e.g.*, linked to itself through another CDO).

"Relationship definition" or "relationship description" in the context of this document and computer software applications refers to information, or an abstraction of information, regarding a "relationship", "data relationship" "relationship type" or a "link" that can be stored, accessed, transferred, communicated, displayed or edited.

"Complex data object graph" or "CDOG" is a term employed herein as an abstraction to logically represent a set of complex data objects and a set of their corresponding relationships.

"Java data object graph" or "JDOG" is a term employed herein as an abstraction to logically represent a set of complex data objects and a set of their corresponding relationships that are part of a Java programming application.

"Application model" or simply "model" are essentially interchangeable terms employed herein as abstractions to logically convey a collective description or other representation for a set of complex data objects and a corresponding description or other representation of their relationships. In one respect, these terms are used logically herein provide a general way of efficiently communicating when referring to set of metadata (i.e., data about data) that describes possible data entities (*e.g.*, objects, database tables,

maps, etc,) data relationship types, and data constraints involved in a computer system or application, or in a specific instance of an application. It is important to understand the context in which the terms "application model" and "model" are used in this document. Ordinarily computer engineers refer to the "model" as an abstraction rather than a specific possibility or instance of the model as applied. However, in this document for the ease of communication abstractions of the model, possible implementations of the model and instances of the model are all referred to generally as "application model" or "model". From the context of its use the term will be clear.

"Navigation", "navigating" or "navigated" in the context of the present document refers to an action implementing at least one object to interact with a set of related objects for a certain purpose, such as creation, access, insertion, modification and deletion of an object, or of one of its relationships.

"Navigation model" as used herein is a special type of application model that is applied specifically to a description (or other representation) of how objects can relate to each other and what might be the expected behavior when a CDOG is navigated for a certain purpose.

"Object schema" is a term employed herein as an abstraction referring to the set of data object classes that describe the possible data objects that can be created, modified or maintained in an application, or describing an instance of a set of data object classes in an application.

"Distributed Transparent Persistence" is a term employed herein as an abstraction referring to the concept of providing persistence for a member selected from the group consisting of a data object, a data object graph, associated data and data object relationships in a distributed environment without the need for the insertion of byte code or data objects in an object model or schema.

"CocoBase Proxy Classes" is a term employed herein used in referring to wrapper classes that provide CocoBase runtime compatibility for objects that aren't inherently database aware. A computer system can persist the attributes and data for any data object that is wrapped with a CocoProxy wrapper class by simply using CocoBase facilities. For example, source code for the (attribute based) CocoProxy and (get/set method based)

CocoProxyM classes are available under the `thought\cocodemo3tier31\demos\pguide` directory, when the CocoBase software tools suite is installed on a computer system.

"CocoBase Navigation API" is a term employed herein to refer to an example of an API that provides database relationship mapping and object graph management capability for persistent objects. Database relationships are mapped to object links using CocoBase Navigator link definitions. Persistence control is provided at each class level in the object graph. Each of the *Select*, *Insert*, *Update* and *Delete* operations are individually configurable.

"CocoBase Transaction API" is a term employed herein to refer to an example of an API that provides object oriented transaction support. Transaction objects are used to persist data object attributes and maintain synchronization between database and in memory attribute values. The Transaction API has many built in optimizations, and applications utilizing CocoBase transactions generally benefit from reduced database and network overhead.

"CocoBase Factories" is a term employed herein to refer to examples of software modules and softwares libraries that are used to provide automated, custom object instantiation behavior. Factory behavior is completely customizable. For example, a factory may be used to bind newly instantiated objects to a transaction object, to load a graph of related objects using the CocoBase Navigator, or to implement polymorphism in a database result set. For example, a ProxyFactory class is part of the current CocoBase software tools suite distribution in the `thought\cocodemo3tier31\demos\pguide` directory, and this factory returns result set objects wrapped in a CocoProxy wrapper, when a CocoProxy wrapped key object is passed into the CocoBase runtime software module as part of a query that needs processing by the CocoBase runtime module.

"CocoBase Repository" is a term employed herein as an abstraction referring to a datasource to dataobject mapping repository and associated software modules that is installed into a datasource (or may optionally be a single stand alone file, or a set of files that circumscribe a set of datasource to dataobject mapping definitions and associated software modules). A repository can optionally be in a format such as XML, XMI and the like. See, U.S. Patent Number 5,857,197, the CocoBaseEnterprise O/R Tools Suite, and the co-pending patent appliction entitled "Dynamic Object-Driven Database

Manipulation and Mapping System" for more detailed descriptions of mapping repositories, and the like.

"CocoBase Transparent Persistence for Objects and Object Models". All models using a relational database for map storage require the CocoBase repository to be installed into the database, or in a stand-alone source accessable to CocoBase. The installation of a mapping repository can occur automatically, if required, when using CocoAdmin to log into the database. Pre-existing database tables can be used, provided that the CocoBase repository is first installed into the database, or accessible to CocoBase. Several example of applications that implement CocoBase transparent persistence are included in the CocoBase software tools suite distribution under the `demos\pguide\navapi` and `demos\pguide\transpersist` directories.

## SUMMARY OF THE INVENTION

An object of the present invention is to provide a system for creating or maintaining transparent persistence of a complex data object, a complex data object graph (CDOG) model, or a portion of a CDOG. In a preferred embodiment, an object of the present invention is to provide such a system that can selectively persist all or a portion of a CDOG model when the model is a member selected from the group consisting of an object model generated from a data object mapping repository and an object model generated from data object modeling tool repository. A further object is to provide such a system is located on, or is part of, a local or distributed computer system.

An object of the present invention is to provide a method for creating, maintaining, accessing, navigating and persisting complex data objects stores in a repository. In a preferred embodiment, an object of the present invention is to provide such a method having the step utilizing the storage facilities of an enterprise EJB server to store and maintain the data object repository. In a further object, such a method involves a local or distributed computer system.

An object of the present invention is to provide a computer software component that operates in an EJB environment, or the like, wherein the component has the capacity to access an object model repository or an instance thereof in a computer memory or in another temportary computer storage store device and persist at least one action selected

from the group consisting of creating, maintaining, accessing, navigating, updating or deleting complex data objects as a CDOG model. In a preferred aspect, the computer software component is an Enterprise Bean selected from the group consisting of Stateless, Stateful and Entity Beans. In a further preferred object the computer software component is an EJB Session Bean built on top of CocoBase runtime libraries having the ability to persist all or a portion of a CDOG model or instance thereof. An even more preferred object is to provide such a computer software component capable of transparently persisting all or a portion of a CDOG model or instance thereo for a local or distributed computer system and automatically reconciling and persisting any changes to an instance of the CDOG model or any changes to the repository definition for the CDOG model.

A preferred object of the present invention is to provide a software tool comprising the a navigation API and softwared component ( as described above), adapted for a local network or a distributed network environment, wherein said software tool provides persistence in an object oriented language environment transparently by implementing a configurable network component capable of acquiring and persisting CDOGs through network APIs.

A further object of the present invention is to a software tool capable of reading a source programming object logic model or a database file in a format selected from the group consisting of a UML data file,m a XMI data file, and a XML file and converting the information into a target member selected from the group consisting of a database definition XML file, a database mapping definition file, and a CDOG definition file. In a preferred object, the software can automatically generate a transparent persistence layer that corresponds to the object model information of the source file.

A further object of the present invention is to provide a software module and source code known as a an Java entity bean (such as a generic session bean) that is capable of providing persistence of either or both of a data objects and a data model, in total or in part as determined through setting established by a user of the computer system.

## BRIEF DESCRIPTION OF THE DRAWINGS

For the non-limiting purpose of illustrating some of the concepts of complex data objects CDOs, *i.e.*, data objects and their relationships to one another, according to the invention, two CDO graph drawings **Figure 1** and **Figure 2** are provided.

**Figure 1** is a complex data object (CDO) graph drawing, which illustrates a customer object and some of its related objects (billing address, orders and items ordered), as well as relationships between the objects. Relationships of the types 1 to 1 (1-1) and 1 to many (1-M) are shown in this CDO graph. More specifically, **FIG. 1** illustrates a CDO graph drawing presenting an instance of a customer object **1** having a 1 to 1 (1-1) relationship **(5)** with its customer billing address object **10,** and a 1 to many relationship (collectively **15, 25,** and **35)** with the three outstanding order objects **20, 30** and **40,** respectively. Order object **20** is an instance of an outstanding order object having a 1 to many relationship (collectively **45** and **55)** with the two items ordered objects **50** and **60,** respectively. Order object **30** is an instance of an outstanding order object having a relationship with a single order item, but order object **30** has a 1 to many relationship **(65)** with the item ordered object **70,** since many order items could have been associated. Order object **40** is an instance illustrates a 1 to many relationship (collectively **75** and **85)** with the two items ordered objects **80** and **90,** respectively.

**Figure 2** is a complex data object (CDO) graph drawing, which illustrates a company object and some of its related objects (corporate address object and some of its departments and employees), as well as relationships between the objects. Relationships of all three types: 1 to 1 (1-1), 1 to many (1-M) and many to many (M-M) are shown in this CDO graph. More specifically, **FIG. 2** illustrates a CDO graph drawing presenting an instance of a company object **100** having a 1 to 1 relationship **(650)** with its corporate address object **700,** and a 1 to many relationship (collectively **150, 250,** and **350)** with the three company department objects **200, 300** and **400,** respectively. Since employees of this company may work for more than one of the company's departments, the company department objects **200, 300** and **400** in **FIG. 2** are three instances (many) of company department objects having relationships (**425, 450, 550** and **575,** respectively) with two (many) employee objects (respectively, **500** and **600**). The cross-assignment of employee object **500** to both company department objects **200** and **300,** and of employee object **600** to both company department objects **300** and **400,** illustrate a complex many to many (M-M) relationship of departments to employees for this company.

## DISCRIPTION THE INVENTION

The present invention provides a system for creating or maintaining persistence for all or a part of a complex data object graph model when a user is creating, maintaining, accessing and navigating complex data objects as a complex data object graph model. In one embodiment, the system comprises a computer system having a user interface, a working memory storage area and at least one device for permanently storing information, and said system being loaded with at least one portion of a computer software program, wherein said software program comprises at least one user access interface and a set of programming routines designed for creating or maintaing transparent persistence when a user is creating, maintaining, accessing and navigating complex data objects as a CDOG model. In a preferred embodiment, the present invention provides such a system that can persist any part or all of a CDOG model instance, and to resolve and persist any changes to the model or to the respository for the CDOG model. Most preferred is such a system providing a point and click graphical user interface.

The present invention provides a method for creating, maintaining, accessing, navigating and persisting complex data objects stores as a CDOG model and providing transparent persistence in a distributed environement, comprising the steps of:

a) creating or accessing a CDOG representation definition or accessing an instance of the CDOG,

b) monitoring and recording any changes to the CDOG or a portion thereof on a local or distributed computer system, and

c) persisting any changes to the CDOG on a local or distributed computer system by updating the CDOG or CDOG representation definition to reflect and changes to any copies of the CDOG or CDOG representation definition, and saving a copy of the updated CDOG representation definition to a data source or to another type of information repository.

In a preferred embodiment, the present invention provides such a method wherein the information repository is an object data store managed by an EBJ Enterprise server. In a further preferred embodiment an EBJ Enterprise Java Bean is the software

component having the ability to access a CDOG repository file, an UML XMI file or an instance of the at least one portion of the CDOG in a computer's memory or on a distributed network and provide the transparent persistence fot the CDOG model.

The present invention provides a computer software program having at least one user interface and having at least one feature that provides for at least one action selected from the group consisting of creating or maintaining transparent persistence when a user of a CDOG model is creating, maintaining, accessing or navigating a CDOG model. In a preferred aspect of the invention, the present invention provides a software program, or an association with a software program, having a feature for displaying, printing or displaying and printing a representation of the CDOG model as a graph, or as a set of tables representing a CDOG model. In a further preferred embodiment, such a software program has an editing interface for editing the CDOG model, or has an editable input or source, such as a file, that can be modified to implement changes to the complex data object CDOG model (or has both an interface for editing and an editable input or source, such as a file).

The present invention also provides an application programming interface ("API"), as a preferred embodiment, wherein the API can be accessed to create, maintain, access, navigate and persisting complex data objects as a CDOG model. In one aspect the API can be accessed by an independent computer software program, or by a computer software program module that is part of a software package including the API, to provide at least one action selected from the group consisting of creating, maintaining, accessing, navigating and persisting complex data objects as a CDOG model. In a preferred aspect, the present invention provides such an API as part of a software tool package that provides a method for displaying or printing a representation of the CDOG model as a graph or set of tables representing the CDOG model (or providing methods for both displaying and printing a representation of the CDOG model). In another preferred aspect, such a software tool package that includes the API provides an editing interface for editing the CDOG model, or has an editable input or source, such as a file, that can be modified to implement changes to the CDOG model (or provides both an editing interface and an editable input or source, such as a file,).

A preferred embodiment of the present invention provides a software tool comprising the API according ( as described above), adapted for a local network or a distributed network environment, wherein said software tool provides persistence in an

object oriented language environment transparently by implementing a configurable network component capable of acquiring and persisting CDOGs through network APIs.

In another preferred embodiment, the CDOG API is the CocoNavigator API which provides support for manipulating complex objects in conjunction with the THOUGHT Inc. CocoBase Enterprise Object/Relational (O/R) database (hereinafter "CocoBase") mapping tool (see U.S. Patent 5,857,197 (incorporated herein in its entirety), for concepts upon which the CocoBase mapping tool is based). An object is considered to be complex when it has established or potential relationships (i.e. links) to other objects. For instance, a Company object instance may be linked to an instance of its corporate address and instances of a Department object, which in turn may be linked to a set of instances of Employee objects. An example of such an CDOG is shown in **FIG. 2**, for example. The combination of these objects and their links constitute a graph of objects (CDOG model) that can be manipulated using the CocoNavigator API. Since the CocoNavigator API works in a Java Programming language environment, this CDOG example may also be referred to as a JDOG example.

In another preferred embodiment, the invention provides a software tool comprising the API according to the invention or interfacing therewith, wherein the software is adapted for performing the following steps:

a) reading a source programming object logic model or a database definition file in a format selected from the group consisting of a Unified Modeling Language ("UML") data file, an XML Metadata Interchange ("XMI") data file, and an Extensible Markup Language ("XMI") file; and
b) converting the information of (a) into a target member selected from the group consisting of a database defintion XML file, a database mapping definition file, and a CDOG definition file.

In a further preferred embodiment, the present invention provides a software tool as described above that is adapted for performing at least one of the following additional steps:

a) displaying a representation of a source programming object logic model or a database definition file in a format selected from the group consisting of a UML data file, a XMI data file, and a XML file;

b) storing a representation of a source programming object logic model or a database definition file in a format selected from the group consisting of a UML data file, a XMI data file, and a XML file;

c) printing a representation of a source programming object logic model or a database definition file in a format selected from the group consisting of a UML data file, a XMI data file, and a XML file;

d) displaying the target information of (b);

e) storing the target information of (b); and

f) printing the target information of (b).

In a still further preferred embodiment the software tool described above provides target information that is at least one member selected from the group consisting a CocoBase database definition repository file, a CocoBase map, a CocoNavigate CDOG definition, a CocoNavigate object link descriptor, and a CDOG object graph definition data file. Further preferred is such software, wherein the source is a UML/XMI document containing sufficient DTD information to exchange modeling information with a UML modeling tool.

In one embodiment of the software tool according to the invention, the software tool can be set to automatically generate a persistence layer that corresponds to the source UML class diagrams provided by an exported source file from a case tool capable of exporting UML/XMI, such as Rational Rose, Together and ArgoUML.

An easy way to understand the job performed by a CDOG Navigator API, such as the CocoNavigator API, is to imagine it as a monitor that can provide at least one object to intercept any accesses to the data objects and their relationships that constitute the CDOG, as well as any changes to the data source upon which the CDOG depends, in order to provide a means to persist any changes to the CDOG (or CDOG model) or related data in the data source. Any changes to the CDOG (or CDOG model) can then be propagated by the CDOG Navigator API to a persistent data source through a CocoBase Enterprise O/R connection. Similarly, persistent data that updates a data source can be utilized to create a CDOG model or to update a CDOG model.

Each CDOG (or CDOG model) managed by the CDOG Navigator API can be associated by the CDOG Navigator API with a CDOG descriptor (such as a file) that may be utilized to represent all or part of a "navigation model". In this respect, a navigation

model may be abstractly described as essentially a namespace in which a set of data objects and a set of link descriptions (*i.e.*, relationship types between data objects) are each defined. In a preferred embodiment, a data source, data field, data field size and data field type are listed for each data object in the set of data objects. In another preferred embodiment, at least one link description for two data objects, or for a single data object having a circular link (where the link description is utilized in conjunction with the CocoNavigator API and CocoBase mapping tool) contains one or more of the following types of information (however other related information may be utilized):

- at least one link type (*e.g.*, 1:1, 1:M, M:1 and M:M) between data objects
- at least one link qualifier (currently supported qualifiers are STD and BEAN)
- at least one associative CocoBase map (optional, for M:M links only)
- the names of the CocoBase maps related to each of the data objects (usually two maps that are not necessarily distinct), and
- information that may be obtained for each of the CocoBase maps related to the relationship between two data objects, including:
  - the name of a relationship link between the two data objects
  - the names of the fields (*i.e.*, keys) in a CocoBase map that are used to establish a relationship link between two data objects
  - the name of a data source for each of the fields (*i.e.*, keys) in a CocoBase map that are used to establish a relationship link between two data objects
  - the names of the fields (*i.e.*, keys) in the associative CocoBase map (optional, for M:M links only)
  - the names of a data source for each of the fields (*i.e.*, keys) in the associative CocoBase map (optional, for M:M links only)
  - the name(s) of any java classes corresponding to fields that would impact upon the relationship between two data objects
  - a setting parameter indicating whether the a loading of a particular CocoBase map that will impact upon a relationship between two data object should be cascaded to other related map(s)
  - a setting parameter indicating whether a deletion of information from a particular CocoBase map that will impact upon a relationship between two data object should be cascaded to other related map(s)

- a setting parameter indicating whether an insertion of information into a particular CocoBase map that will impact upon a relationship between two data object should be cascaded to other related map(s), and
- a setting parameter indicating whether an update of information in a particular CocoBase map that will impact upon a relationship between two data object should be cascaded to other related map(s).

Some preferred features provided by the CocoNavigator API and its associated software, when utilized with the CocoBase mapping tool, or with a similar mapping tool, are as follows:

(a) *provides multiple navigation models for the same CDOG model*

A preferred embodiment of the CocoNavigator API or an associated program module is configured to allow many navigation models to be used with the same set of java classes and CocoBase maps that are associated with a CDOG model. This preferred embodiment API, or an associated computer program module, may also be configured to permit a user to switch dynamically from one such navigation model to another. Thus, while a given application is being executed, it is possible to completely change the relationships between data objects and how CDOGs of a CDOG model should be managed.

(b) *circular link definitions*

A preferred embodiment of the CocoNavigator API, or an associated computer program module, is configured to permit a user to create, access, support and correctly manage circular data object links. In a navigation model, a circular link may be defined as occurring when a data object is directly or indirectly linked to itself as part of a CDOG. Such relationships can be created, accessed, supported and correctly managed via the CocoNavigator API.

(c) *bi-directional and oriented links*

A preferred embodiment of the CocoNavigator API, or an associated computer program module, is configured to permit a user to create, access,

support and correctly manage links between data objects as either a bi-directional link or as an oriented link. In this embodiment links between data objects are usually bi-directional links and can be navigated back and forth as an un-oriented navigational model of the CDOG model. With such a bi-directional link feature activated, any object in the navigation model in a given CDOG model can be used as a navigation entry point. In an oriented link navigation model, the link may be set as an oriented link accessible from a single direction. Thus, changes must be made from top down or from bottom up on a relationship tree representation of the navigation model, and some actions may need to originate from a pre-set node of the relationship tree. In this embodiment, such types of links can be created, accessed, supported and correctly managed via the CocoNavigator API.

(d) *dynamic link proxy properties*

In a preferred embodiment, the CocoNavigator API, or an associated computer program module, is configured to permit a user to populate any object property (*e.g.*, public field or a getter/setter pair) having its type declared as an object class with special dynamic proxies that can monitor and update the state of a relationship link. Examples of such types of object classes are thought.Cocobase.navapi.LinkObjProxy of CocoBase and the Sun Microsystems java class, java.util.Vector. Bi-directional or single directional object references can be consistently maintained in this manner, or by other similar logical mechanisms.

(e) *automatic synchronization of object properties*

In a preferred embodiment, the CocoNavigator API, or an associated computer program module, can be configured to automatically merge a data object and its properties into a CDOG navigation model representation when the class of object being navigated (being created, accessed, supported or maintained) has a property (*e.g.*, a field or a getter/setter pair) with a name matching the name of a corresponding link as defined in the navigation model.

(f) *pure object models*

In a preferred embodiment, the CocoNavigator API, or an associated computer program module, can be configured to include a data object as a relationship link to another data object in a navigation model without the need to declare fields and properties for the data object. Such links should be limited however to cases where no computer code dependency will exist between the java classes that will be associated with such linked data objects of the navigation model. Thus, such a navigation model can be a pure abstraction, and more reusable than just a populated version or single hypothetical instance of the abstraction.

(g) *customized link visitors*

In a preferred embodiment, the CocoNavigator API, or an associated computer program module, can provide an interface which can be configured to permit a visitor object to use this interface to visit (*e.g.*, access or change) a data object that is part of a CDOG navigation model. In conjunction with CocoBase, an example of CocoNavigator API implementing this concept might using a LinkVisitor object (class). Other similar classes may be defined for this purpose. The visitor interface provides a way for a user to customize and adapt the software, and thereby allows a user to extend the functionality of the CocoNavigator API, or the functionality of an associated computer program module, to provide a desired customizable behavior in their CDOG navigation model.

(h) *distributed environments*

In a preferred embodiment, the CocoNavigator API, or an associated computer program module, can be configured to operate as a tool to create, access, support and correctly manage a CDOG navigation model in a server environment (*e.g.*, in an EJB container) and to persist any changes to the CDOG navigation model when a navigation model is distributed across a local network or when the navigation model involves a distributed network (*e.g.*, a navigation model distributed across internet connections). In one aspect, on the server side of the network, a CDO or a CDOG model of any complexity sent by clients (*e.g.*, serialized copies) across a local network, or across a distributed network, can be correctly merged into a

CDOG model by the CocoNavigator API, or by an associated computer program module. Additionally, the CocoNavigator API, or an associated computer program module, can be configured to send a CDO or CDOG model to a client along with link proxies serialized with parts of the CDO or CDOG model that are being monitored by such link proxies. By sending the CDO or CDOG model copy along with such link proxies and associated parts of the CDOG or CDOG model to a client, dynamic link proxies (described above in (d)) can be used by the client side. Accordingly, a CDOG navigation model can be created, accessed, supported, managed and persisted over a distributed network.

(i) *group loading or lazy loading of links*

In a preferred embodiment, the CocoNavigator API, or an associated computer program module, can be configured to monitor some or all of the data objects (and associated relationship definitions) participating in links of a CDOG navigation model. In a more preferred embodiment, the data objects (and associated relationship definitions) participating in links of a CDOG navigation model can be loaded as they are needed (lazy loading) from a persistent data source or from another type of information repository. This lazy loading feature can permit a very large CDOG navigation model to be loaded in a per-link basis as needed. In a more preferred embodiment, the lazy loading feature can be configured to prevent a link from being loaded twice and can be configured to detect cycles of related loads that have been previously loaded. Such configuration adaptability can be exploited to provide the more efficient use of the resources for a computer system or network or for a distributed computer network.

Some examples of preferred features that can be provided by the CocoNavigator API and its associated software, which are particularly enhanced when utilized with the currently existing CocoBase mapping tool, are as follows:

(aa) *CocoBase transactions*

A preferred embodiment of the CocoNavigator API or an associated program module is configured to bind a data object or data object

relationship of a CDOG navigation model with a CocoBase transaction. Such configurations may be utilized to optimize access to the database server and to enhance performance of the computer system, or systems, involved with the CDOG navigation model.

(bb) *non-CBProp objects*

The CocoNavigator API or an associated program module can be configured to automatically detect non-CBProp objects (objects with classes that do not implement the CocoBase interface known as the CBProp interface), and automatically create proxies in order to persist such objects as part of a CDOG navigation model.

(cc) *virtual foreign key fields*

The CocoNavigator API or an associated computer program module can be configured to use proxy classes of CocoBase, such as the CocoProxyM classes, when appropriate. Implementing proxy classes such as the CocoProxyM classes can provide a system for creating, accessing, supporting, properly maintaining and persisting virtual foreign key fields (*i.e.*, foreign key fields do not need to be defined in the object class for Java programming implementations) by the CocoNavigator API, or by an associated computer program module.

(dd) *transparent persistence*

The CocoNavigator API or an associated computer program module can be configured to use a configurable network component capable of acquiring and persisting CDOGs through network APIs and thereby provide persistence transparently with respect to the applications that are using the CDOGs. In a preferred implementation, there is provided a software tool comprising the CocoNavigator API that is adapted for a local network or for a distributed network environment, wherein the software tool provides persistence in an object oriented language environment transparently by implementing a configurable network component capable of acquiring and persisting CDOGS. Such a concept may be referred to as transparent persistence in the context of this API, because persistence can be obtained without requiring reconfiguring of an

existing CDOG model or software application in order to provide such persistence to a computer software application.

From the above description of features of the API, and the features of the CocoNavigator API programming routines described below, an object computer language programmer can produce an API having the functionality of the CocoNavigator API described herein. Essentially the CocoNavigator API has an at least one user access interface, at least one data source access interface and at least three main programming modules consisting of the following programming modules or routines:

I. A programming module or routine constructed to operate on a computer system and to provide the following features to a user or to a software program module of the computer system:
   a) a computer programming sub-routine or sub-module for obtaining from a member selected from the group consisting of a data source, an information repository, and an input device, sufficient information to construct a CDOG model containing one or more CDOGs,
   b) a computer programming sub-routine or sub-module for constructing and loading into the memory registers of the computer system a CDOG or a CDOG model representation definition from a data source or other repository, and
   c) a computer programming sub-routine or sub-module for sending a copy of some portion or all of the CDOG representation definition to a user or to a software program module on a local computer system or on a distributed network;

II. A programming module or routine constructed to operate on a computer system and to provide the following features to a user or to a software program module of the computer system:
   a) a computer programming sub-routine or sub-module for monitoring a user or a software program module on a computer system that has accessed or changed a portion of a CDOG or CDOG model, which is included in the CDOG, or CDOG model, representation definition of (I), above, and obtaining any changes to the CDOG or CDOG model,
   b) a computer programming sub-routine or sub-module for monitoring a user or a software program module on a computer system who has

obtained a copy of any portion of the CDOG, or CDOG model, representation definition, and for obtaining a copy of any changes that the user might have made to any portion of the CDOG, or CDOG model, representation definition, and

c) a computer programming sub-routine or sub-module for comparing a copy of a CDOG, CDOG model, or a representation definition of either the CDOG or CDOG model, to an original stored version of the CDOG, CDOG model, or an original stored representation definition for the CDOG or CDOG model, and for updating the original to incorporate any changes to a CDOG or a representation definition that are made by the user or by a software program module; and

III. A programming module or routine constructed to operate on a computer system and to provide the following features to a user or software program module of the computer system:

a) a computer programming sub-routine or sub-module for storing a new CDOG or CDOG model, or storing a definition of either the CDOG or CDOG model, in a data source or in another information repository, and

b) a computer programming sub-routine or sub-module for persisting (saving to permanent storage) either a changed portion of an updated CDOG, an updated CDOG model, or an updated definition representation for either a CDOG or a CDOG model, to a data source or to another type of information repository.

In a more preferred embodiment, the CDOG API according to the invention, (a preferred embodiment is the CocoNavigator API) can be written with an interface that accesses and uses functionalities of the CocoBase mapping tool, associated programming routines, or associated class libraries. Both the object code and the source code of the CDOG API are contemplated as part of this invention.

In another preferred embodiment, the above described CDOG API can be accessed to create, maintain, access, navigate and persisting complex data objects as a CDOG model. In a particularly preferred aspect, the API can be accessed by a computer software program such as the CocoBase mapping tool, or by a computer software program module that is part of a software package including the API, to provide at least one action selected from the group consisting of creating, maintaining, accessing, navigating and persisting complex data objects as a CDOG model. In an even more

preferred aspect, the present invention provides such an API as part of a software tool package having a feature or method for displaying or printing a graph or set of tables representing a CDOG or a CDOG model. In yet another preferred aspect, such a software tool package including the API provides an interface for editing a CDOG, a CDOG model, or for editing a representation definition for a CDOG or a CDOG model to provide a modification that implements a change to a CDOG, or to its definition.

As described above, a software component according to the invention that is capable of persisting all or a portion of a CDOG may be an Enterprise Java Bean selected from the group consisting of Stateless, Stateful or even Entity Beans. CocoBase Enterprise Beans (CBEBs) are a preferred embodiment of the invention anc constitute a set of Enterprise Java Beans build upon or accessing CocoBase Runtime and EBJ APIs to provide persistence in a generic, efficient and transparent way. Distributed transparent persistence can be provided without the need for any object model changes or byte code modifications. CBEBs can be directly deployed on an EJB server with or without customized settings and are capable of persisting virtually any Java Object, any Java Object graph or any portion of the Object graph. The Java Objects of any complexity that are arbitrarily created by client applications can be persisted, and the corresponding object classes can be properly mapped to a CocoBase repository or other object repository such as a UML XMI repository. Copies of the Java Object, any portion of an Object Graph, or all of an Object Graph can be automatically reconciled and the changes can be persisted to a storage repository.

The CocoBase Navigation API maps database *foreign key – primary key* relationships to object references. The relationship maps (sometimes referred to herein as navigation models), like the database maps, are stored in a special CocoBase repository, or may be converted into an XML file (or files) and stored in a stand-alone file or directory. The Navigation API supports 1 to 1, 1 to M, and M to M links in both the object and relational spaces. For M to M relationships, an associative table and a corresponding associative map are required, since relational databases do not directly support M to M relationships.

A navigation model must be created, or properly reference, before CocoBase links can be defined. A navigation model is a *namespace* used to categorize and store a set of link definitions. Generally, a single navigation model is sufficient for simple applications. Multiple navigation models are useful if an application switches between several

relationship configurations. This allows different views of complex object data for the same set of underlying database tables. Having relationship mapping separated from table mapping is unique to CocoBase. Since maps can be used with multiple link models, it provides reusability at the mapping level.

Below is an example of how a client application on an EJB Server could connect to and use at least on CBEB to provide transparent persistenct, even in a distributed environment.

***Example 1***

```
            // create a complex object having myObj as the root of the tree
            ...
            Context initialContext = new InitialContext();
            SfCmbHome cmbHome =
(SfCmbHome)javax.rmi.PortableRemoteObject.narrow(
                    initialContext.lookup("java:comp/env/ejb/sfcmbhome"),
            SfCmbHome.class);
            SfCmb myCmb = SfCmbHome.create(.../*connection details*/...);

            // create an example object
            MyObject myObj = new MyObject();
            myObj.setSomeField(someValue);

            // now ask the CMB to load the object from the database that matches
the example
            myObj = myCmb.retrieveState(myObj);

            myCmb.saveState(myObj);
            ...
            // do some more changes to myComplexObject and save it again
            ...
            myCmb.saveState(myObj);
            ...
```

***Implemention of Examples 2-4, below, collectively provide persistence via a session bean as described generally by the following text.***

The CBSession EJB

------------------

The CBSession EJB is a Stateful Session Bean that works as an abstraction layer for CocoBase runtime classes. It provides generic transparent persistence to Java objects of any complexity in a simple yet elegant way. It is designed to run in any Java Application Server, thus combining CocoBase transparent object persistence capabilities with most features provided by such servers, such as transaction management, security, resource pooling and so forth.

The CBSession bean is very simple and easy to use. Here is an illustration of how the session bean of Examples 2-4, below might be implemented and used:

```
...
Context ic = new InitialContext(...);
Object ref = ic.lookup("CBSession");
CBSessionHome home =
        (CBSessionHome) PortableRemoteObject.narrow
(ref,CBSessionHome.class);

String cocourl =
"CBSession;cocorep=configresource:/resources/MyConfigFile.properties:cocoprop=prim
arykey.lookup=false,dynamic.querying=true,jndiurl=java:comp/env/jdbc/demoPool"

// connects to a CBSession
CBSession cbSession = null;
try {
        cbSession = home.create(cocourl,"MyNavModel",null);
        cbSession.setDebugLevel(1);
} catch (Exception e) {
        System.out.println("Could not open CBSession due to " + e);
        return;
```

```
}

MyComplexObject obj = new MyComplexObject();
...
// populate values for querying
obj.setValues(...);
...
// load object graph
obj = (MyComplexObject) cbSession.load(obj,"MyMap");
...
// do some changes to obj graph
...
// save changes
cbSession.save(obj,"MyMap");
...
```

See CBSession interface for more details.

Deploying the CBSession bean
----------------------------

Requirements:

- coco_home/classes and coco_home/demos must be in your App Server classpath.
- Ant 1.2 or later must be installed in order to compile/regenerate deployable jar files.

The Ant script build.xml will generate a deployable jar file with the CBSession bean for a particular server. Make sure the entry in the beginning of the build.xml file has the entry

<property name="cocohome"  value="c:/thought/cocodemo3tier31" />

correctly specified.

Targets currently defined include WebLogic 6.1, JBoss 2.4 and a generic jar file. Each subdirectory corresponds to a server, containing the standard descriptor and

proprietary descriptors for that server. These descriptors can be modified in case it is required to do so.

For example, the jar file for JBoss can be generated as follows:

```
os-prompt> ant jboss
```

See build.xml for available targets.

IMPORTANT: In order to have CBSession working for a particular application model, all the required java classes and resources (i.e. the map pre-load configuration properties file, the navigation model properties file) must be in the classpath (for example, these classes and resources could be copied to the coco_home/demos/resources directory, provided this directory is included in the app server classpath). Another alternative would be to add all needed classes and resources to the CBSession bean jar file. This can be done simply by copying these classes and resources to the directory corresponding to the target server and running the ant script again. For example, for JBoss, the following directory structure would generate a jar file with all classes and resources:

```
cbsession
    ...

    jboss
        ejb-jar.xml
        jboss.xml
        resources
            MyConfigFile.properties
            MyNavModel.properties
        my
            app
                package
                    AppClass1.class
                    AppClass2.class
                    AppClass3.class
                    ...
```

Examples 2-4, source code for CBSession Bean and its use are as follows:

***Example 2***

Source Code File Name = CBSession.java

```
package thought.CocoBase.ejbs.cbsession;

import java.rmi.RemoteException;
import javax.ejb.EJBObject;

/**
 * CBSession remote interface
 * <p>
 * Copyright (c) 2001 THOUGHT Inc. All Rights Reserved.
 *
 * @author Alexandre Jonatan Martins
 * @version 0.9, August 2001
 */
public interface CBSession extends EJBObject {

        /**
         * Invokes <code>load(Object,String)</code> sending
         * <code>object.getClass().getName()</code> as the
<code>mapName</code>
         * @param obj the example object
         * @return the loaded object
         */
        public Object load(Object obj) throws RemoteException;

        /**
         * Loads an object from the database
         * @param obj the <i>example</i> object
         * @param mapName the map name corresponding to the class of obj
         * @return the loaded object
```

```
    */
    public Object load(Object obj, String mapName) throws
RemoteException;

    /**
     * Invokes <code>save(Object,String)</code> sending
     * <code>object.getClass().getName()</code> as the
<code>mapName</code>.
     * @param obj the example object
     */
    public void save(Object obj) throws RemoteException;

    /**
     * Saves an object to the database
     * @param obj the <i>example</i> object
     * @param mapName the map name corresponding to the class of obj
     * @return the loaded object
     */
    public void save(Object obj, String mapName) throws RemoteException;

    /**
     * Invokes <code>delete(Object,String)</code> sending
     * <code>object.getClass().getName()</code> as the
<code>mapName</code>.
     * @param obj the example object
     */
    public void delete(Object obj) throws RemoteException;

    /**
     * Deletes an object from the database
     * @param obj the <i>example</i> object
     * @param mapName the map name corresponding to the class of obj
     */
    public void delete(Object obj, String mapName) throws RemoteException;

    /**
```

```
     * Returns app server session debug level
     * @return <code>true</code> if debug is on; <code>false</code>
otherwise
     */
    public int getDebugLevel() throws RemoteException;

    /**
     * Sets app server session debug level as follows:
     * <p><code><pre>
     *    1 - prints basic session response
     *    2 - prints navigation info + level 1 info
     *    3 - prints detailed navigation info + level 2 info
     *    4 - prints cocobase transaction info + level 3 info
     *    5 - prints cocobase driver and jdbc info + level 4 info
     * </pre></code></p>
     * @param debug the debug level
     * @param mapName the map name corresponding to the class of obj
     */
    public void setDebugLevel(int level) throws RemoteException;

}
```

***<u>Example 3</u>***

Source Code File Name = CBSessionHome.java

```
package thought.CocoBase.ejbs.cbsession;

import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EJBHome;

import java.util.Properties;

/**
 * CBSession home interface
```

```
 * <p>
 * Copyright (c) 2001 THOUGHT Inc. All Rights Reserved.
 *
 * @author Alexandre Jonatan Martins
 * @version 0.9, August 2001
 */
public interface CBSessionHome extends EJBHome {

    /**
     * Creates a CBSession. Properties will be read from the evironment
entries in
     * the ejb descriptor. The following properties are recognized:
     * <p><code><pre>
     * cocosource.url = cocobase_connection_url_string
     * cocosource.name = cocobase_driver_name_string
     * cocosource.user = user_name_string
     * cocosource.password = user_password_string
     * cocosource.autoclose =  "true" | "false"
     * cocosource.autotrans =  "true" | "false"
     * cocosource.navmodel = navigation_model_string
     * </pre></code></p>
     */
    public CBSession create() throws RemoteException, CreateException;

    /**
     * Creates a CBSession. The behavior of the session data source can be
customized by
     * a properties object containing specific configuration details. The
following
     * properties are recognized:
     * <p><code><pre>
     * cocosource.name = cocobase_driver_name_string
     * cocosource.user = user_name_string
     * cocosource.password = user_password_string
     * cocosource.autoclose =  "true" | "false"
     * cocosource.autotrans =  "true" | "false"
```

```
                * </pre></code></p>
                * @param cbUrl the CocoBase connection url
                * @param navModel the name of the navigation model (if not provided,
uses default navigation model)
                * @param props configuration properties
                */
                public CBSession create(String cbUrl, String navModel, Properties props)
throws RemoteException, CreateException;
        }
```

***Example 4***

Source Code File Name = CBSessionBean.java

```
        package thought.CocoBase.ejbs.cbsession;

        import java.util.Enumeration;
        import java.util.Properties;
        import java.rmi.RemoteException;
        import javax.ejb.CreateException;
        import javax.naming.InitialContext;
        import javax.naming.Context;
        import javax.ejb.SessionBean;
        import javax.ejb.RemoveException;
        import javax.ejb.SessionContext;
        import thought.CocoBase.CocoDriver;
        import thought.CocoBase.CocoProxyM;
        import thought.CocoBase.CBProp;
        import thought.CocoBase.CocoDriverInterface;
        import thought.CocoBase.CocoFactoryInterface;
        import thought.CocoBase.CocoFactoryDefineInterface;
        import thought.CocoBase.navapi.Navigator;
        import thought.CocoBase.navapi.LinkLoaderInterface;
```

```
/**
 * This is the ejb bean class for the CBSession ejb stateful session bean
 * <p>
 * Copyright (c) 2001 THOUGHT Inc. All Rights Reserved.
 *
 * @author Alexandre Jonatan Martins
 * @version 0.91, January 2002
 */
public class CBSessionBean implements SessionBean {

        /**
         * This flag determines if we auto close the db connection after
         * every database related call in the bean. This should typically
         * be set to true since any EJB servers require this for connection
         * pooling and scaling management. If this is set to false, the
         * connection will only be closed on ejbPassivate, so each bean
         * will retain an open connection to the database while it is active.
         */
        protected boolean autoCloseFlag = true;

        /**
         * This CocoBase Transaction object is used in the bean exclusively
         * to provide update buffering. It is generated only when
         * transaction management is requested during code generation.
         * The purpose of this object is to track changes to non-bean
         * instances, and commit those changes in the ejbStore method.
         * It leaves all 'actual' transaction issues to the Container as per
         * the EJB specification via its 'commitconnection=false' flag.
         */
        protected thought.CocoBase.Transaction cocoTxn = null;

        /**
         * Info for Database, Connections managed by the Container and OTM.
         */
        protected CocoDriverInterface dataSource = null;
```

```
/**
 * the navigator that manages the graph of objects
 */
protected transient Navigator nav = null;

/**
 * the reference to the session context
 */
protected SessionContext sessionContext;

/**
 * the reference to the initial context
 */
protected transient Context initialContext = null;

private boolean _DEBUG = false;
private int debugLevel = 0;

private LinkLoaderInterface localLoader;

public CBSessionBean() {
        if (_DEBUG) System.out.println("CBSession(" +this.hashCode() +
") - contructor invoked");
}

/*=========== remote interface ==========*/

public Object load(Object obj) throws RemoteException {
        return this.load(obj,obj.getClass().getName());
}

public Object load(Object obj, String mapName) throws RemoteException
{
        if (_DEBUG) System.out.println("CBSession(" +this.hashCode() +
") - load invoked; obj = " + obj + " map = " + mapName);
        try {
```

```
                        this.openDataSource();
                        return this.loadLinks(obj,mapName);
                } catch (Exception e) {
                        throw new RemoteException("CBSession Error: load failed
with exception " + e,e);
                } finally {
                        if (this.autoCloseFlag) {
                                this.closeDataSource();
                        }
                }
        }

        public void save(Object obj) throws RemoteException {
                this.save(obj,obj.getClass().getName());
        }

        public void save(Object obj, String mapName) throws RemoteException {
                if (_DEBUG) System.out.println("CBSession(" +this.hashCode() +
") - save invoked; obj = " + obj + " map = " + mapName);
                try {
                        this.openDataSource();
                        if (this.nav.findObject(obj,mapName) == null) {
                                // we make sure to have the links loaded for the
state instance
                                this.loadLinks(obj,mapName);
                        }
                        this.nav.updateAllLinks(obj,mapName,true); // update all
links
                        cocoTxn.commit(); // this should commit changes done to
the navigator instance!
                } catch (Exception e) {
                        throw new RemoteException("CBSession Error: save failed
with exception " + e,e);
                } finally {
                        if (this.autoCloseFlag) {
                                this.closeDataSource();
```

```
				}
			}
		}

		public void delete(Object obj) throws RemoteException {
			this.delete(obj,obj.getClass().getName());
		}

		public void delete(Object obj, String mapName) throws RemoteException
{
			if (_DEBUG) System.out.println("CBSession(" +this.hashCode() +
") - delete invoked; obj = " + obj + " map = " + mapName);
			try {
				this.openDataSource();
				if (this.nav.findObject(obj,mapName) == null) {
					// we make sure to have the links loaded for the
state instance
					this.loadLinks(obj,mapName);
				}
				this.nav.deleteAllLinks(obj,mapName,true); // delete all
links
				cocoTxn.commit(); // this should commit changes done to
the navigator instance!
			} catch (Exception e) {
				throw new RemoteException("CBSession Error: delete
failed with exception " + e,e);
			} finally {
				if (this.autoCloseFlag) {
					this.closeDataSource();
				}
			}
		}

		public void setDebugLevel(int level) throws RemoteException {
			this.debugLevel = level;
			this._DEBUG = this.debugLevel > 0;
```

```
			if (this.nav != null) {
				this.nav.setDebugLevel(this.debugLevel - 1);
			}
			if (this.cocoTxn != null) {
				this.cocoTxn.setDebug(this.debugLevel >= 4);
			}
			if (this.dataSource != null) {
				this.dataSource.setDebug(this.debugLevel >= 5);
			}
		}

		public int getDebugLevel() throws RemoteException {
			return this.debugLevel;
		}

		/*============ private section ============*/

		/**
		 * Loads the state instance from the database and its links.
		 * Assumes there is a valid connection between the data source
		 * and the navigator.
		 */
		private Object loadLinks(Object obj, String mapName) {
			if (_DEBUG) System.out.println("CBSession(" +this.hashCode() +
") - loading links...");
			// load the matching instance from the datasource
			Object navInstance = this.nav.findObject(obj,mapName);
			if (navInstance != null) {  // if obj has a corresponding instace in
the graph
				if (_DEBUG) System.out.println("CBSession("
+this.hashCode() + ") - instance graph already loaded; returning loaded instance");
				return navInstance; // return instance in the graph
			}
			Object dbobj = null;
			if (obj instanceof CBProp) {
				dbobj = this.dataSource.select(obj,mapName);
```

```
			} else {
				dbobj = this.dataSource.select(new
CocoProxyM(obj),mapName);
			}
			if (dbobj == null) { // object is not in the database,; return null;
				if (_DEBUG) System.out.println("CBSession("
+this.hashCode() + ") - object is not in the database; returning null");
				return null;
			}
			Object loadedObj = this.nav.loadAllLinks(dbobj,mapName,true);
// load all links
			if (loadedObj != null) {		// root is not automatically bound to
transaction
				this.cocoTxn.bind(loadedObj);	// so we have to explicitly
bind it
			}
			if (loadedObj instanceof CocoProxyM) { // if loaded instance is
wrapped by a proxy
				loadedObj = ((CocoProxyM)loadedObj).getObject(); //
unwrap it
			}
			if (_DEBUG) System.out.println("CBSession(" +this.hashCode() +
") - links loaded!");
			return loadedObj; // and return loaded instance
		}

		/**
		 * This method checks and closes the non-null dataSource - database
		 * connection and sets the dataSource variable to null.
		 */
		private void closeDataSource() {
			try {
				// Make sure connection isn't null before closing.
				if(dataSource != null) {
					if (_DEBUG) System.out.println("CBSession("
+this.hashCode() + ") - data source closed");
```

```
                this.dataSource.close();
            }
        } catch (Exception e) {
        }
        this.dataSource = null;
    }

    /**
     * Establishes a connection to our datasource if one doesn't exist,
     * otherwise a new instance is created!
     *
     * @exception RemoteException  Thrown if the instance could not
perform
     *             the function requested by the container because of a
     *             system-level error.
     */
    private void openDataSource() throws RemoteException{
        if (this.dataSource == null) {
            // we may need to get properties from the environment, so
let's get a handle to the Context
            Context envContext = null;
            try {
                this.initialContext = new InitialContext();
            } catch (Exception re) {
                throw new
RemoteException("CBSession.openDataSource: cannot get InitialContext", re);
            }
            try {
                envContext =
(Context)initialContext.lookup("java:comp/env");
            } catch (Exception re) {
                throw new
RemoteException("CBSession.openDataSource: cannot get environment context", re);
            }
            try {
                if (this.prop_url == null) {
```

```
                                        this.prop_url =
(String)envContext.lookup("cocosource.url");
                                }
                        } catch (Exception re) {
                                throw new
RemoteException("CBSession.openDataSource: cannot get cocosource.url from
context",re);
                        }
                        if (this.prop_url == null || this.prop_url.length() == 0) {
                                throw new
RemoteException("CBSession.openDataSource: cocobase url missing!\n\tPlease specify
the environment entry cocosource.url in your descriptor");
                        }

                        try {
                                if (this.prop_driver == null) {
                                        this.prop_driver =
(String)envContext.lookup("cocosource.name");
                                        if (this.prop_driver == null ||
this.prop_driver.length() == 0) {
                                                this.prop_driver =
"thought.CocoBase.CocoPowderPlugin20";
                                        }
                                }
                        } catch (Exception re) {
                                this.prop_driver =
"thought.CocoBase.CocoPowderPlugin20";
                                if (_DEBUG) {

            System.out.println("CBSession.openDataSource: cannot get
cocosource.name from context due to " + re);
                                        System.out.println("Assuming
cocosource.name = " + this.prop_driver);
                                }
                        }
```

```
                    try {
                            if (this.prop_user == null) {
                                    this.prop_user =
(String)envContext.lookup("cocosource.user");
                                    if (this.prop_user == null ||
this.prop_user.length() == 0) {
                                            this.prop_user = "";
                                    }
                            }
                    } catch (Exception re) {
                            this.prop_user = "";
                            if (_DEBUG) {

            System.out.println("CBSession.openDataSource: cannot get
cocosource.user from context due to " + re);
                                    System.out.println("Assuming
cocosource.user = " + this.prop_user);
                            }
                    }

                    try {
                            if (this.prop_password == null) {
                                    this.prop_password =
(String)envContext.lookup("cocosource.password");
                                    if (this.prop_password == null ||
this.prop_password.length() == 0) {
                                            this.prop_password = "";
                                    }
                            }
                    } catch (Exception re) {
                            this.prop_password = "";
                            if (_DEBUG) {

            System.out.println("CBSession.openDataSource: cannot get
cocosource.password from context due to " + re);
```

```
                                        System.out.println("Assuming
cocosource.password = " + this.prop_password);
                                }
                        }

                        try {
                                if (this.prop_autoTransactions == null) {
                                        this.prop_autoTransactions =
(String)envContext.lookup("cocosource.autotrans");
                                        if (this.prop_autoTransactions == null ||
this.prop_autoTransactions.length() == 0) {
                                                this.prop_autoTransactions = "false";
                                        }
                                }
                        } catch (Exception re) {
                                this.prop_autoTransactions = "false";
                                if (_DEBUG) {

                System.out.println("CBSession.openDataSource: cannot get
cocosource.autotrans from context due to " + re);
                                        System.out.println("Assuming
cocosource.autotrans = " + this.prop_autoTransactions);
                                }
                        }

                        try {
                                if (this.prop_autoClose == null) {
                                        this.prop_autoClose =
(String)envContext.lookup("cocosource.autoclose");
                                        if (this.prop_autoClose == null ||
this.prop_autoClose.length() == 0) {
                                                this.prop_autoClose = "true";
                                        }
                                }
                        } catch (Exception re) {
                                this.prop_autoClose = "true";
```

```
if (_DEBUG) {

System.out.println("CBSession.openDataSource: cannot get
cocosource.autoclose from context due to " + re);
System.out.println("Assuming
cocosource.autoclose = " + this.prop_autoClose);
}
}

try {
if (this.prop_navModel == null) {
this.prop_navModel =
(String)envContext.lookup("cocosource.navmodel");
if (this.prop_navModel == null ||
this.prop_navModel.length() == 0) {
this.prop_navModel = "";
}
}
} catch (Exception re) {
this.prop_navModel = "";
if (_DEBUG) {

System.out.println("CBSession.openDataSource: cannot get
cocosource.navmodel from context due to " + re);
System.out.println("Assuming
cocosource.navmodel = " + this.prop_navModel);
}
}
try {
if (_DEBUG) {
System.out.println("CBSession("
+this.hashCode() + ") - connecting to data source...");
System.out.println("\tProperty settings:");
System.out.println("\tcocosource.name =
" + this.prop_driver);
```

```
                        System.out.println("\tcocosource.url      = "
+ this.prop_url);
                        System.out.println("\tcocosource.user:    = "
+ this.prop_user);
                        System.out.println("\tcocosource.password
= " + this.prop_password);
                        System.out.println("\tcocosource.autotrans =
" + this.prop_autoTransactions);
                        System.out.println("\tcocosource.autoclose
= " + this.prop_autoClose);
                        System.out.println("\tcocosource.navmodel
= " + this.prop_navModel);
                    }
                    this.dataSource =
CocoDriver.getCocoDriver(this.prop_driver,"",this.prop_url,this.prop_user,this.prop_pas
sword);
                    if(this.dataSource instanceof
thought.CocoBase.CocoPowder) {

        thought.CocoBase.CocoPowder.setSilentFlag(true);
                    }
                    this.dataSource.setThrowExceptions(true);
                    this.dataSource.setDebug(this.debugLevel >= 5);
                    this.dataSource.setThis(this);
                    CocoFactoryInterface proxyFactory = new
CocoFactoryInterface() {
                        public Object getInstance(Object src, Object
props, String objectName) {
                            try {
                                Object myObj = null;
                                // We want to handle proxy
objects with special care here.
                                if(src instanceof
thought.CocoBase.CocoProxyM) {
                                    // Create a new
instance of proxy class
```

```
                                                    myObj =
src.getClass().newInstance();
                                                    // Create a new
instance of contained class
                                                    Object
newContainedObject =
((thought.CocoBase.CocoProxyM)src).getObject().getClass().newInstance();
                                                    // Set myObj to
contain newContainedObject

          ((thought.CocoBase.CocoProxyM)myObj).setObject(newContainedObject
);
                                               } else {
                                                    // Create an instance
of non-proxy, CBProp class
                                                    myObj =
src.getClass().newInstance();
                                               }
                                               // Populate the objects
attributes
                                               if(myObj instanceof CBProp)

          ((CBProp)myObj).setPropObjectData((Properties)props);
                                               // Return instance of CBProp
class
                                               return myObj;
                                          } catch (Exception e) {
                                               e.printStackTrace();
                                          }
                                          return null;
                                     }
                                };
                                // comment out this line if a proxy factory is not
required

          ((CocoFactoryDefineInterface)this.dataSource).setFactory(proxyFactory);
```

```
                    this.dataSource.connect();
                    if(this.prop_autoClose != null &&
this.prop_autoClose.equalsIgnoreCase("false"))
                        this.autoCloseFlag = false;
                    else
                        this.autoCloseFlag = true;
                    if(this.prop_autoTransactions != null &&
this.prop_autoTransactions.equalsIgnoreCase("true")) {
                        this.dataSource.setAutoCommit(true);
                        if (_DEBUG)
        System.out.println("CBSession(" +this.hashCode() + ") - transactions
disabled; presuming non-OTS compatible source!");
                    }

                } catch (Exception e) {
                    throw new
RemoteException("CBSession.openDataSource: cannot connect to " + this.prop_driver,
e);
                }
            }
            if (this.cocoTxn == null) {
                if (_DEBUG)  System.out.println("CBSession("
+this.hashCode() + ") - creating CocoBase transaction");
                this.cocoTxn = new thought.CocoBase.Transaction(false);
                Properties cocoTransProps = new Properties();
                cocoTransProps.put("preserveCommit","true");
                cocoTransProps.put("commitconnection","false");
                cocoTransProps.put("throwExceptions","true");
                this.cocoTxn.setProperties(cocoTransProps);
                this.cocoTxn.setDebug(this.debugLevel >= 4);
                this.cocoTxn.begin();
            }
            cocoTxn.setDatabase(dataSource);

            if (this.nav == null) {
```

```
                        if (_DEBUG)  System.out.println("CBSession("
+this.hashCode() + ") - creating CocoBase navigator");
                        Properties navProps = new Properties();
                        navProps.setProperty("useCurrentTransaction","false");
                        navProps.setProperty("alwaysCreateProxies","true");
                        this.nav.setDebugLevel(this.debugLevel - 1);
                        this.nav = new Navigator(this.prop_navModel);
                        this.nav.setClassForLoader(this.getClass());
                        this.nav.setProperties(navProps);
                }
                this.nav.setConnection(this.dataSource);
                this.nav.setTransaction(this.cocoTxn);
                // this is where we set it up for distributed lazy load

        this.nav.setLinkLoader((CBSession)this.sessionContext.getEJBObject());
        }

        // this is for distributed lazy loading only
        public Object loadLink(Object object, String mapName, String linkName)
throws RemoteException {
                if (_DEBUG) System.out.println("CBSession(" +this.hashCode() +
") - loadLink called - distributed lazy load in use!");
                try {
                        this.openDataSource();
                        return
this.nav.loadLink(object,mapName,linkName,true,true,true);
                } catch (Exception e) {
                        throw new RemoteException("CBSession Error: loadlink
failed with exception " + e,e);
                } finally {
                        if (this.autoCloseFlag) {
                                this.closeDataSource();
                        }
                }
        }
```

```
/*============ javax.ejb.EntityBean implementation ==========*/

public void ejbCreate() throws RemoteException, CreateException {
        if (_DEBUG) System.out.println("CBSession(" +this.hashCode() +
") - session created. Properties will be read from descriptor.");
}

public void ejbCreate(String cbUrl, String navModel, Properties props)
throws RemoteException, CreateException {
        if (_DEBUG) System.out.println("CBSession(" +this.hashCode() +
") - session created. Setting properties...");
        this.prop_url = cbUrl;
        this.prop_navModel = navModel;
        if (props != null) {
                Enumeration keys = props.keys();
                while (keys.hasMoreElements()) {
                        Object key = keys.nextElement();
                        Object value = props.get(key);
                        if ("cocosource.name".equals(key)) {
                                this.prop_driver = (String) value;
                        } else if ("cocosource.user".equals(key)) {
                                this.prop_user = (String)value;
                        } else if ("cocosource.password".equals(key)) {
                                this.prop_password = (String)value;
                        } else if ("cocosource.autotrans".equals(key)) {
                                this.prop_autoTransactions = (String)value;
                        } else if ("cocosource.autoclose".equals(key)) {
                                this.prop_autoClose = (String)value;
                        }
                }
        }
}

public void ejbActivate() throws RemoteException {
        if (_DEBUG) System.out.println("CBSession(" +this.hashCode() +
") - session activated");
```

```
		}

		public void setSessionContext(SessionContext context) throws
RemoteException {
			if (_DEBUG) System.out.println("CBSession(" +this.hashCode() +
") - setting session context");
			this.sessionContext = context;
		}

		public void ejbPassivate() throws RemoteException {
			//we close our connection
			closeDataSource();
			this.nav = null; // we reset all navigation info so that it won't be
serialized
			this.cocoTxn = null; // we do the same for the trasaction
			if (_DEBUG) System.out.println("CBSession(" +this.hashCode() +
") - session passivated");
		}

		public void ejbRemove() throws RemoteException {
			// we close our connection
			closeDataSource();
			if (_DEBUG) System.out.println("CBSession(" +this.hashCode() +
") - session removed");
		}

		// private properties
		// cocosource.autoclose =  "true" | "false"
		private String prop_autoClose = null;
		// cocosource.autotrans =  "true" | "false"
		private String prop_autoTransactions = null;
		// cocosource.name = coco_driver_name
		private String prop_driver = "thought.CocoBase.CocoPowderPlugin20";
		// cocosource.navmodel = cocobase_navigation_model
		private String prop_navModel = null;
		// cocosource.password = "true" | "false"
```

```
        private String prop_password = null;
        // cocosource.url = cocobase_url
        private String prop_url = null;
        // cocosource.user = "true" | "false"
        private String prop_user = null;

}
```

The present may be embodied in specific forms other than those particularly described above or illustrated by the appended drawings. Upon viewing the present application preferred embodiments and other descriptions herein of the present invention, variations and other implementations that do not depart from the spirit and scope of the present invention will be apparent to one of routine skill in this field. Such variations and other implementations are considered part of the present invention and within the scope of the appended claims. Accordingly, reference should be made to the appended claims, rather than to the forgoing specification and drawings, as indicating the scope of the present invention.

**Appendix 1: From UML models to Transparent Persistence using CocoBase Enterprise OR**

## CocoBase UML/XMI Import Tool

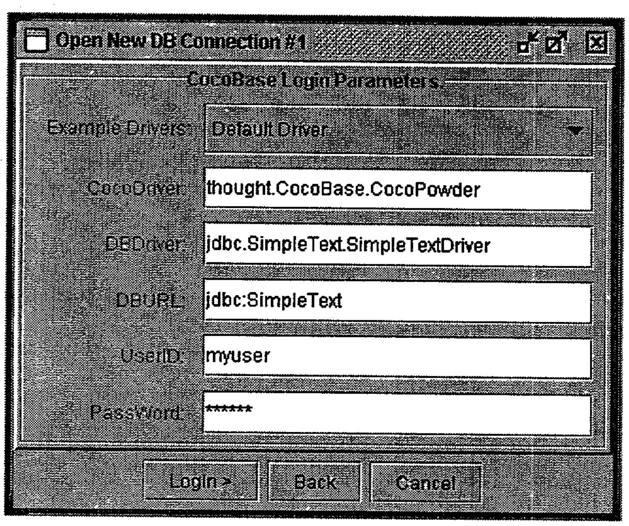
THOUGHT INC CocoBase Enterprise OR provides a tool that imports UML1.3 XMI documents and converts the model information into CocoBase Maps and Link (i.e. Relationship) descriptors. XMI stands for XML Metadata Interchange and it is basically a standard that allows different vendors to exchange modeling information. The XMI standard specifies a Document Type Definition (DTD) for UML so that UML models created with case tools can be converted to/from a XML document.

CocoBase UML/XMI import tool currently supports XMI 1.0/UML 1.3 documents. Thus, CocoBase can automatically generate a complete implementation of a persistence layer that correspond to UML class diagrams created with case tools capable of exporting XML/XMI diagrams (Rational Rose, Together and ArgoUML are known to support it).

The steps to import a UML/XMI model into CocoBase are detailed below.

*1) Connect to the CocoBase Repository*

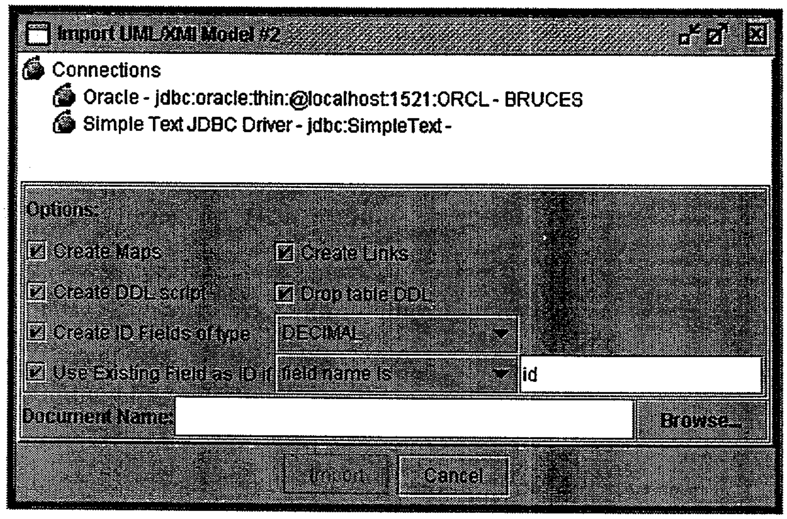
Launch CocoBase and go to **File->Open New DB Connection**
The following screen will appear:




Select the JDBC driver/user/password to be used to connect to the CocoBase repository and click on the **Login >** button.

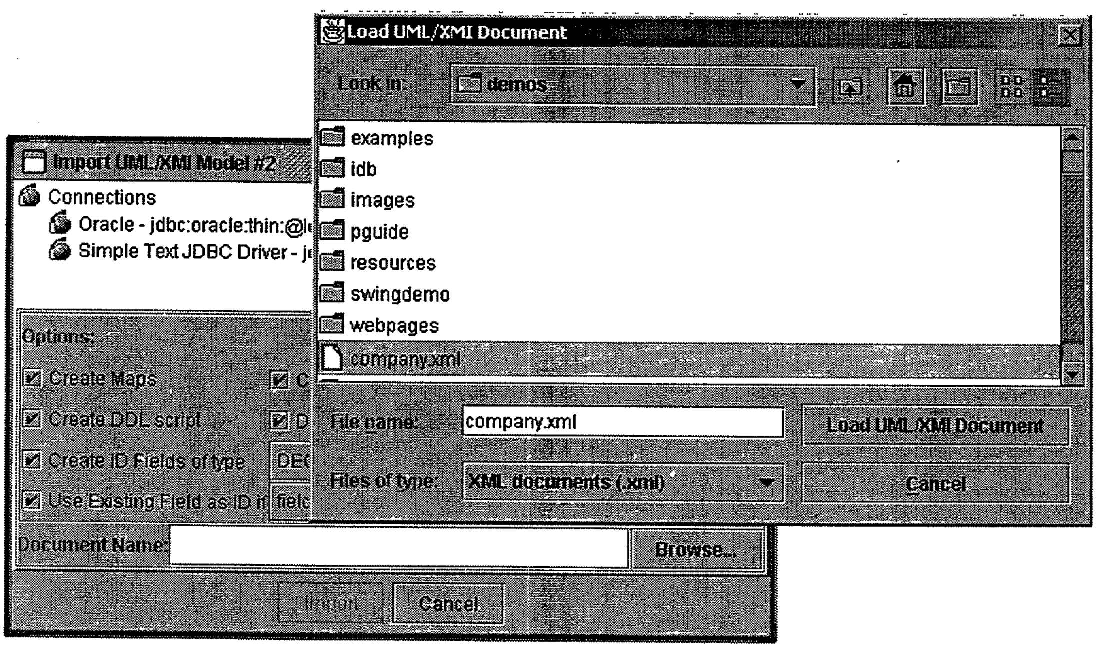
*2) Lauch the UML/XMI Import Tool*

To launch the UML/XMI Import Tool, go to **Tools->Import UML/XMI Model**
A screen like the following should appear:



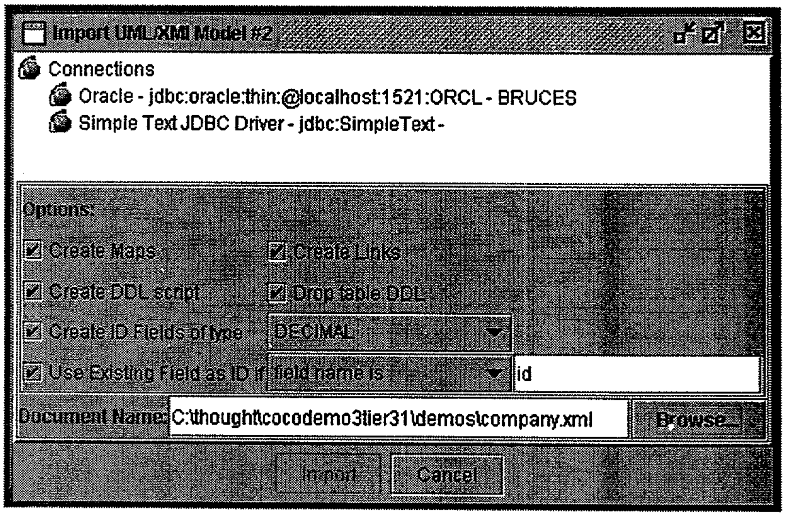

*3) Select a UML/XMI file*

Click on the button **Browse...** to choose a file containing the UML/XMI document to be imported by CocoBase. Make sure the document is compatible with XMI version 1.0 and UML 1.3 type (umlx13.dtd).



*4) Configure Import Options*

The UML/XMI Import Tool has some options that can be configured prior to importation, as described below.




- Create Maps

  Indicates if CocoBase Maps should be created for the classes defined in the UML/XMI document.

- Create Links

  Indicates if CocoBase Link definitions (i.e. relationships) should be created for the associaitons defined in the UML/XMI document.

- Create DDL script

  Indicates if a SQL/DDL script to create the tables according to the generated maps should be created in the database after importation.

- Drop table DDL

  Indicates if DROP TABLE statements should be inserted in the beginning of the DDL script.

- Create ID Fields of type *jdbc_type*

  When this options is checked, a field *ID* will be created in each map and used as the identifier so that it is guaranteed that correct Link definitions will be generated for the associations defined in the UML/XMI document (since all classes will

have an identifier, it is always possible to properly specify foreign-primary key field references in the resultant Link definitions). The type of the created field will conform to the type selected in the drop down box. Note the an ID field will **not** be created in the map if *Using Existing Field as ID* is checked and a existing field is found in the corresponding UML class according to the search criteria specified (see details next).

- Use Existing Field as ID

  Indicates if classes defined in the UML/XMI document should be searched for an identifier field. The search is performed according to the following criteria:

  - field name is *text*:

    searches for an field that has the exact name *text*

  - field name contains *text*:

    searches for an field that has a name containing *text*

  - field name starts with *text*:

    searches for an field that has a name starting with *text*

  - field name ends with *text*:

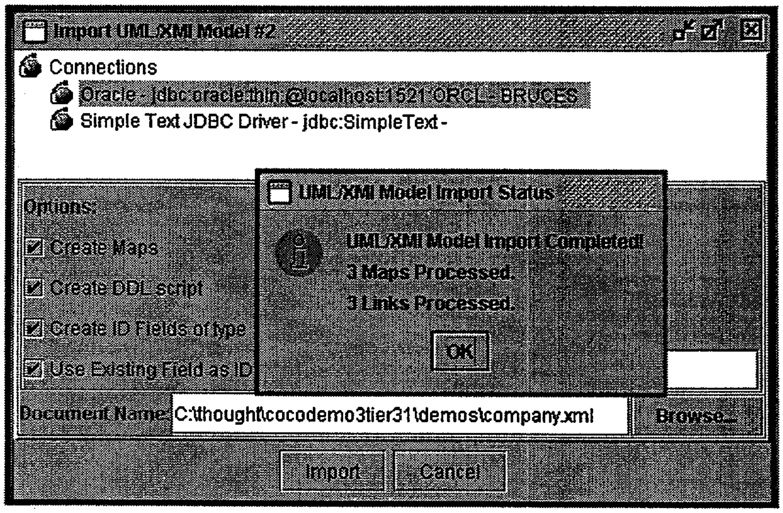
    searches for an field that has a name ending with *text*

  Note that the search for *text* is **case sensitive**. The search also takes inheritance relationships into account, which means the inherited fields are included in the search of a class. When two fields match the criteria in a given class, only one will be picked as the identifier. It is not possible to determine which field will be chosen in this case, since UML/XMI documents may not obey the ordering of the original class diagram. A good practice is to use a special naming convention for identifier fields when designing the model. It is important to point out that, if the Create ID Fields option is checked and a existing field is found and, an ID field will **not** be created, which means that pre-existing fields will have priority over identifiers that might be created.

*5) Import UML/XMI document*

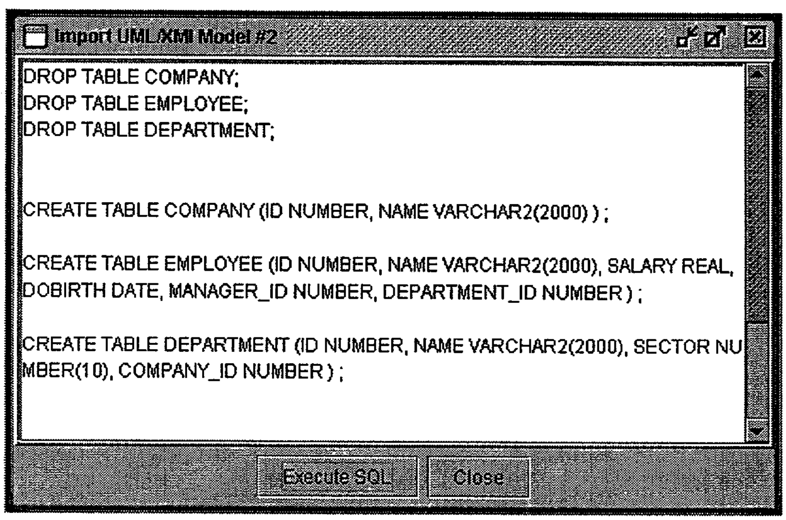
Before importing the UML/XMI document, select a connection to the CocoBase repository in the connection tree. Once a connection is selected and import options are appropriately checked, click on the button **Import** to proceed with the importation of the document.

In case any problem or inconsistency is detected while trying to import the model, a warning dialog will be presented containing the details. When importation is completed, a message indicating the number of CocoBase Maps and Links created will be prompted as shown below:



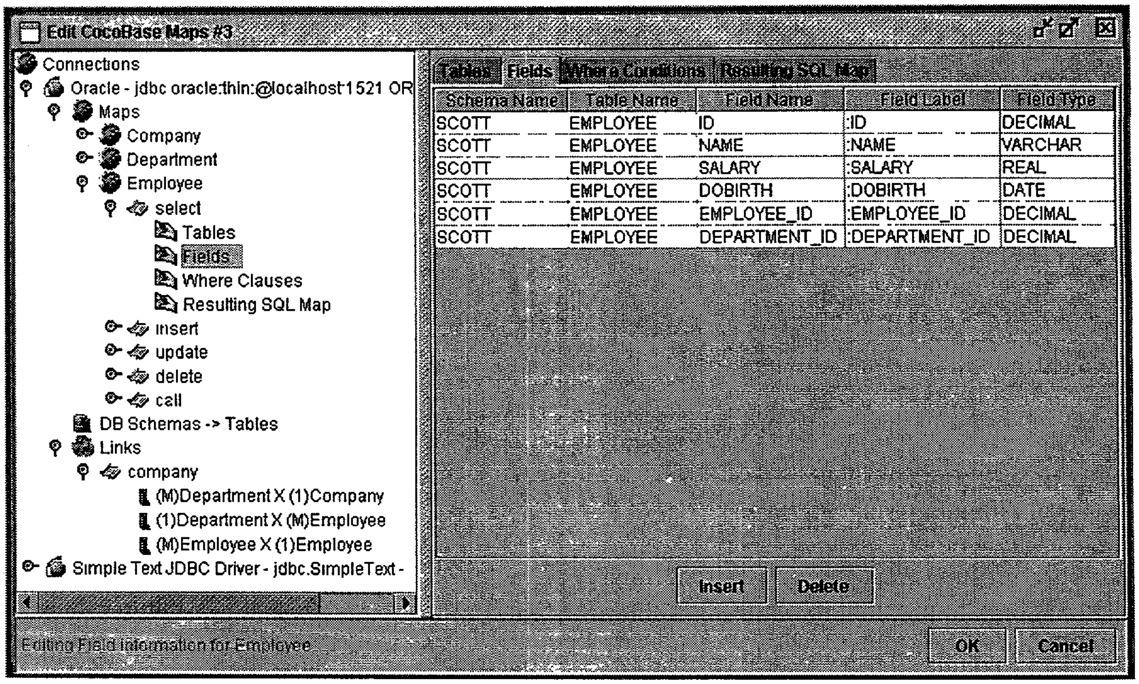

*6) Customize and Execute SQL/DDL script*

If the option Create DDL Script is checked, another screen containing a SQL/DDL script for table creation is presented. It is possible to edit, cut or paste the generated script and even submit it to the database server. To execute the script, click on the button **Execute**. At this point, CocoBase Maps and Links are already created and it the execution of the script can be skipped by clicking on the button **Close**.




*7) Verify the results*

Go to File->Edit Map to open the Map editor and verify if the Maps and Links were created accordingly.




### CocoBase Transparent Persistence

In the previos section, it was demonstrated how to create Maps and Links from a UML model using the UML/XMI Import Tool. In this section, it is shown how CocoBase runtime classes can be used to provide transparent persistence to a Java Object Model which corresponds to the generated Maps and Links.

## Transparent Persistence with CocoBase

CocoBase accomplishes transparent persistence with java object models without using bytecode manipulation, proprietary interfaces or class hierarchy intrusion. This means that no special classes or interfaces are needed in the Object Model in order to do persistence with CocoBase. The only requirement is that they must have a default constructor with no arguments.

There are 3 basic Runtime components that are involved in the transparent persistence of objects:

- The CocoBase Runtime O/R mapping class that wrappers the JDBC driver and issues queries and does the actual persistence calls. This is a class such as `thought.CocoBase.CocoPowder` or `thought.CocoBase.CocoPowderPlugin20` (for jdbc 2.0 connections).

- The `thought.CocoBase.Transaction` object that can track changes of instances, and acts as a change 'buffer'. If a Transaction object is used, then it only calls a CocoBase Runtime driver - O/R mapping runtime class - when the txn.commit() is called.

- The `thought.CocoBase.navapi.Navigator` object that can track and detect changes in relationships based on Link definitions. Note that unlike CocoBase Maps, Link models are not kept in the CocoBase repository. Once a link definition model is created it is saved in the demos/resources directory (by default - although this can be overridden). As long as the model is in the classpath either directly or in a subdirectory called 'resources' it will find the model properties file and retrieve the navigation information.

The `Navigator` class can function in conjunction with a `Transaction` object or it can function standalone. While complex object graphs can be transparently managed directly by the Navigator and without the `Transaction` object, the use of the `Transaction` object is generally preferred because of its buffering and update optimizations which only persist those attributes that have changed.

# Creating Applications with CocoBase Transparent Persistence

Once Links and Maps are properly created, go to **File->Generate Java Code** to generate Java classes. The 'Default CocoNavigate Java Object' code generation target will generate a pure java class with no CocoBase interfaces.

After classes have been generated and compiled, CocoBase runtime classes can be used to persist instances of these classes. First, open a CocoBase connection as follows:

```
CocoDriverInterface myBase = CocoDriver.getCocoDriver(
    "thought.CocoBase.CocoPowder",
    "org.hsql.jdbcDriver",
    "jdbc:HypersonicSQL:hsql://localhost;cocoprop=cocofactory=CocoProxyFactory",
    "sa", ""
)
if(myBase.connect() == -1) {
    System.out.println("Failed connect!");
    System.exit(1);
}
...
```

Now create a CocoBase transaction object to manage any objects that are retrieved. Notice how the transaction object is configured through parameters and property settings.

```
thought.CocoBase.Transaction cocoTxn =
                              new thought.CocoBase.Transaction(myBase, false);
Properties cocoTransProps = new Properties();
cocoTransProps.put("preserveCommit","true");
cocoTransProps.put("commitconnection","true");
cocoTransProps.put("throwExceptions","true");
cocoTransProps.put("updateOnlyChangedColumns","true");
cocoTxn.setProperties(cocoTransProps);

// Begin a new transaction.
cocoTxn.begin();
...
```

Then open a CocoBase Navigator object with the Navigation model to be used and also we register the Transaction object with the Navigation model:

```
    // Instantiate a Navigator with the Link model created from
    // the UML/XMI document
    thought.CocoBase.navapi.Navigator navigator =
                                  new thought.CocoBase.navapi.Navigator (myBase,
"company");
    // Assign the current transaction to the Navigator
    navigator.setTransaction(cocoTxn);
```

Now select the top level node to work with. Notice our use of a CocoProxyM class which 'wrappers' the pure java object model class and gives the class the compatibility with CocoBase through java reflection instead of requiring any special interfaces in the java class itself:

```
    // Setup our object to query
    Department dept = new Department();
    department.setName("SALES");

    // This will read & bind all 'Department' objects to the transaction.
    Vector deptVector = myBase.selectAll(

                                  new
thought.CocoBase.CocoProxyM(dept),"Department");
```

Now it is possible to step through each of these objects and tell the retrieved object to be navigated through the `loadAllLinks` method which does the automatic navigation for that object:

```
    for(int i=0; i< deptVector.size(); i++) {
                                  Department d =
(Department)deptVector.elementAt(i);
                                  // Because the cascadeLoad flag is set to true in
the direction
                                  // Department->Employees, the employees link will
load automatically
                                  d = navigator.loadAllLinks(d,"Department");
                                  ...
                                  Vector emps = d.getEmployees();
                                  for (int j=0; j<emps.size(); j++) {
```

```
        // raise salaries by 20%
        emp.setSalary(emp.getSalary()*1.2);
        ...
                        }
                        ...
    // Once changes are made to an object graph those changes can be
    // synchronized using the updateAllLinks method such as:

                        navigator.updateAllLinks(d,"Department",true);
}
```

You can then commit the buffered changes with:

```
cocoTxn.commit();
```

These code introductions are quite small, and can be done entirely server side with Entity or Session beans in J2EE environments with no model or object model intrusion. And for local non-J2EE applications the application intrusion is incredibly small, requiring a single method call for each root node.

The Navigator supports one-to-one, one-to-many and many-to-many relationships with cycle detection. It also detects this locally (i.e. in the client application) or by reconciling serialized or copied objects without any kind object model or bytecode intrusion. This is truly transparent persistence that is architected and designed for the VM oriented Java language. There are also more advanced applications included in the demos/pguide/navapi subdirectory that demonstrate one-to-one, one-to-many and many-to-many relationships as well as an EJB using the Navigator system to manage a graph of java objects.

**Conclusion**

This document described how to add CocoBase Transparent Persistence to Object Models and applications from a UML diagram exported as an XMI file. It is only a brief overview of what can be done with CocoBase. The architecture and implementation presented here is uniquely suited to work in every app from the tiny local app to the enterprise J2EE and to do so with superior performance and manageability.

Hope this gets you started and if you have any more questions you can post them to the forum at http://forum.thoughtinc.com or by sending support email to support@thoughtinc.com.

## Appendix 2: From SQL to Transparent Persistence using CocoBase Enterprise OR

This document is a brief overview of how to create Java applications that use CocoBase Transparent Persistence features starting from SQL database tables.

## Table Mapping

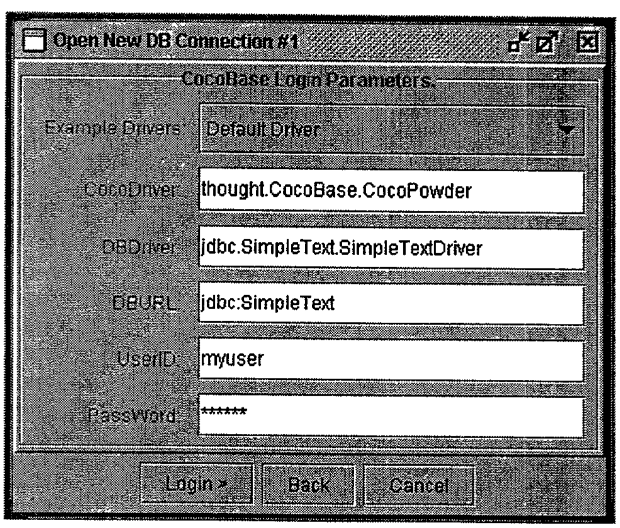
Suppose a set of database tables was defined as follows:

```
CREATE TABLE COMPANY(
                              ID NUMERIC,
                              NAME VARCHAR(50),
                              PRIMARY KEY(ID))
CREATE TABLE DEPARTMENT(
                              ID NUMERIC,
                              NAME VARCHAR(50),
                              COMPANY NUMERIC,
                              PRIMARY KEY(ID),
                              FOREIGN KEY(COMPANY)REFERENCES COMPANY(ID))
CREATE TABLE EMPLOYEE(
                              ID NUMERIC,
                              NAME VARCHAR(50),
                              SALARY REAL,
                              DEPARTMENT NUMERIC,
                              MANAGER NUMERIC,
                              PRIMARY KEY(ID),
                              FOREIGN KEY(DEPARTMENT)REFERENCES DEPARTMENT(ID))
```

The steps to create a CocoBase Map from a given table are detailed below.

*1) Connect to the CocoBase Repository*

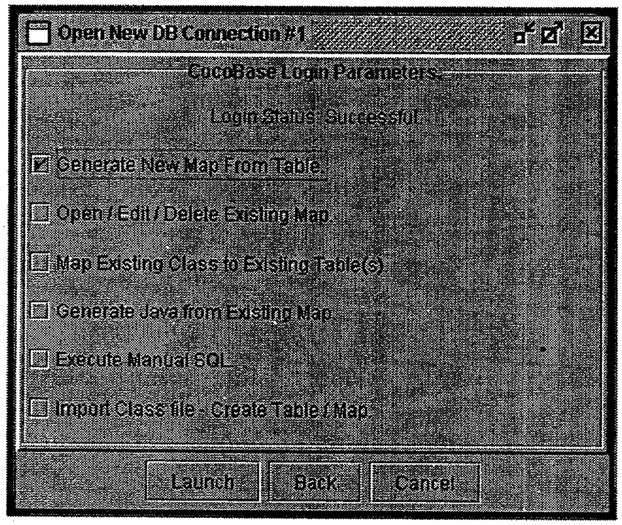
Launch CocoBase and go to **File->Open New DB Connection** to open the login window.



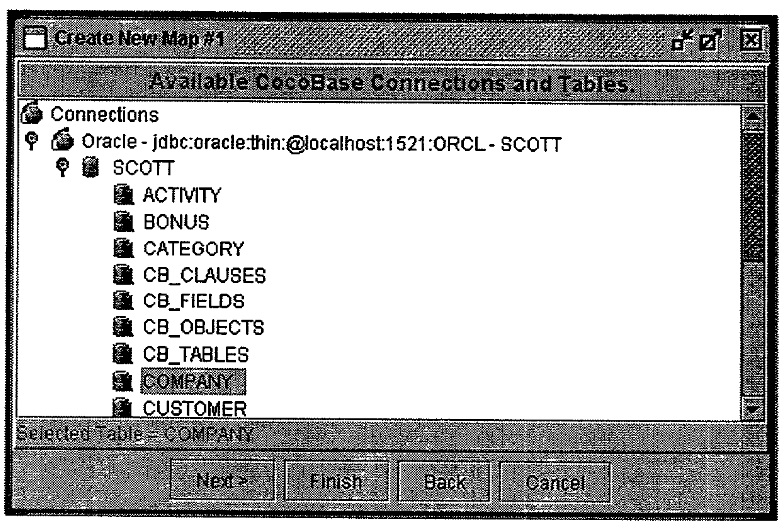
Select the JDBC driver/user/password to be used to connect to the CocoBase repository and click on the **Login >** button.

*2) Create Map from Table*

If login was successful, a window like the following will appear:

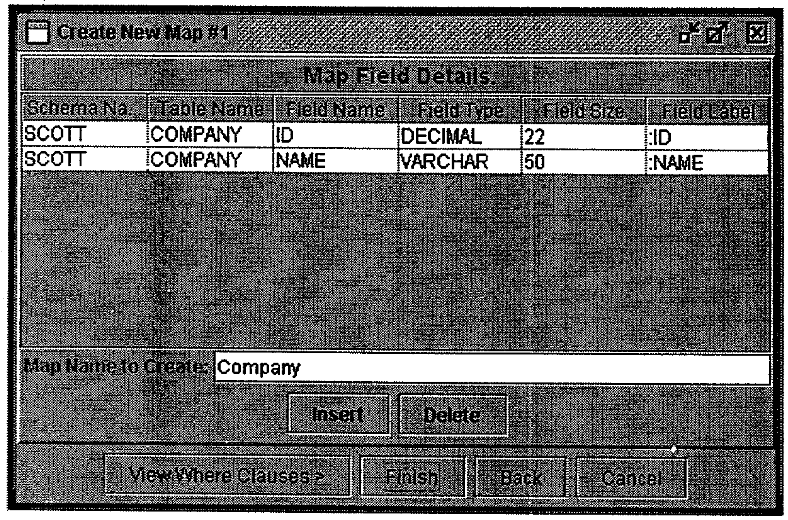



Check the box **Generate New Map From Table** and click on the button **Launch**.
A window similar to the following should appear:

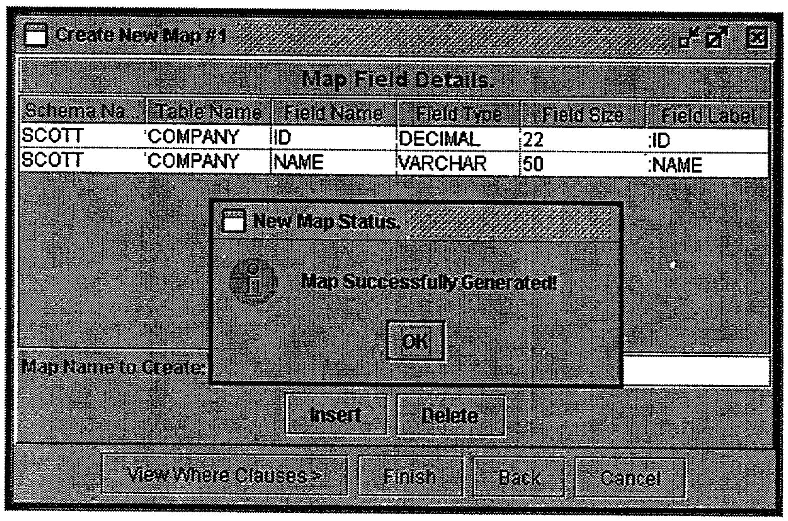



Expand the branches on the tree to see the tables available in the database and select one of the tables. Click on the button **Next >** to go to the window with the Map field details e verify if the information is correctly matching the table definition and if the Map name is

appropriate. Here is possible to modify the Map by changing the names or types of the fields and inserting or deleting fields.



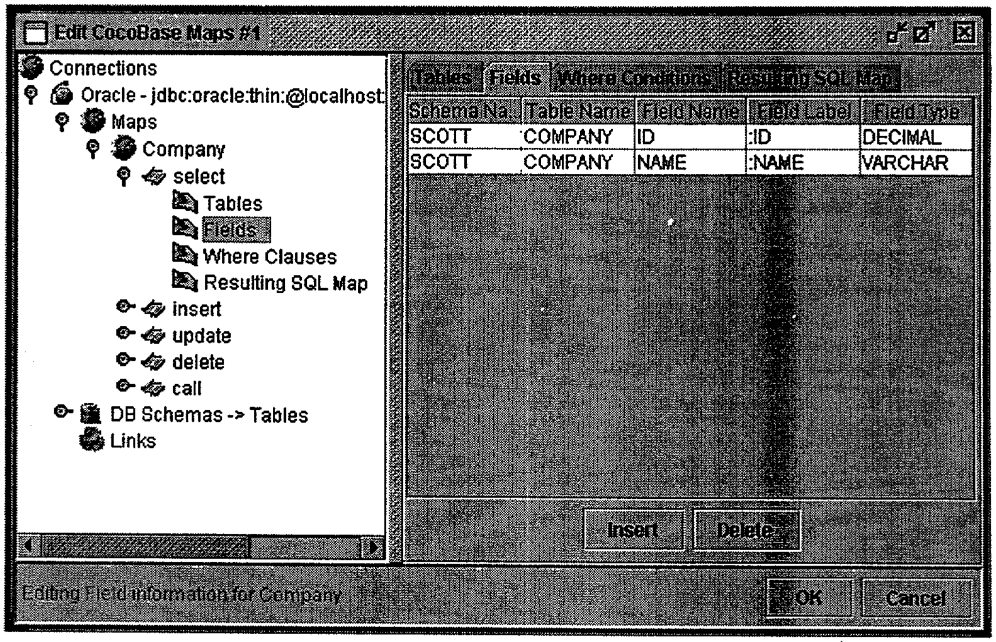

Click on the button **Finish** to create the new Map from the selected table. A dialog will be prompted asking to confirm the creation of the Map. Click on the button **Continue Map Create** to confirm the creation of the new Map. A message will be prompted confirming that the Map was successfully generated and is already in the CocoBase repository as below:




Click on the button Ok. This procedure must be repeated for all the tables containing information relevant to your application. To start again in the Map creation Window, select the menu item
**File -> New Map from Table.**

*3) Verify the results*

Go to **File->Edit Map** to open the Map editor and verify if the Maps were created accordingly.



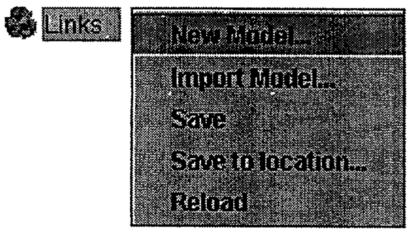

## Relationship Mapping

Once all necessary database tables are mapped, it is possible to create Link definitions between the Maps so that 1-1, 1-M and M-M relationships between objects can be transparently managed by CocoBase. The steps to create a Link definition are detailed below.

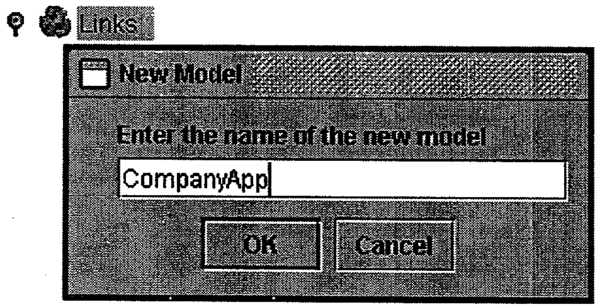
*1) Choose Link Model*

Before creating a Link, it is necessary to have a Link model created. A Link model is a set of Link definitions. Multiple Link models are useful when an application needs to switch between several relationship configurations, since it allows different views of complex object data over the same set of underlying tables. The idea of having relationship mapping separated from table mapping is unique of CocoBase. It provides reusability at the mapping level, since a given Map can be reused across multiple Link models.

Usually, one single Link model is sufficient for a simple application. To create a link model, right click on the **Links** branch in the Map editor tree. A pop-up menu like the following will appear:



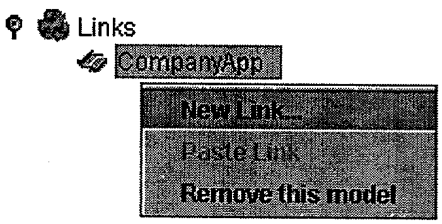

Select the option **New Model...** and enter the name of the new Model in the input dialog that will appear next.



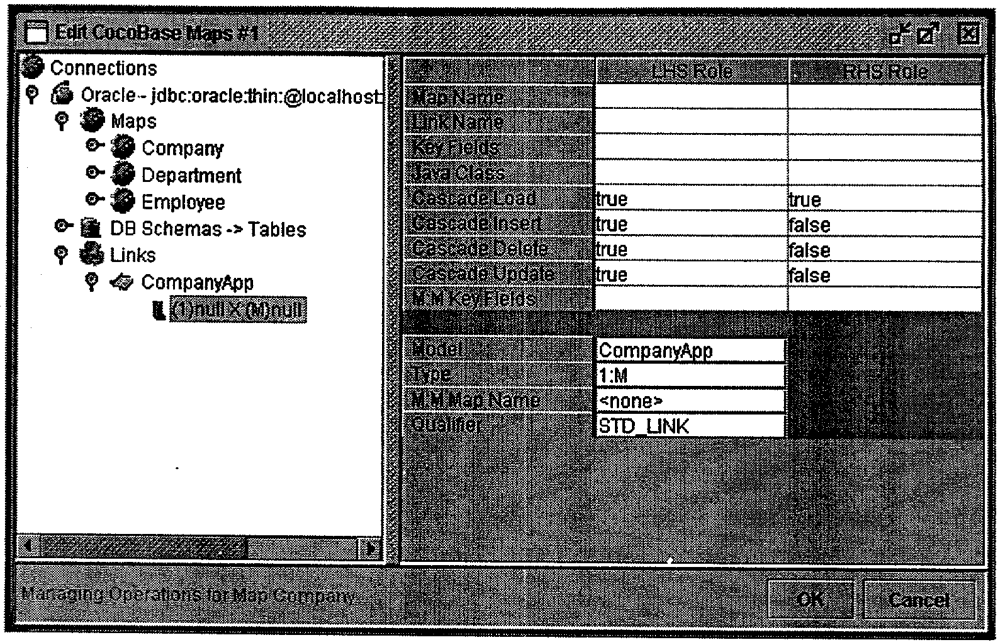

The new Link model will be inserted as a sub-node in the editor tree.

*2) Create Link Definition*

In order to create a new Link for that model, right click on the model to activate the model pop-up menu and select the menu item **New Link....**




After clicking on the menu, a new empty Link definition will be inserted in the selected model. Expand the model branch and click on the new Link definition to see its details, as below:



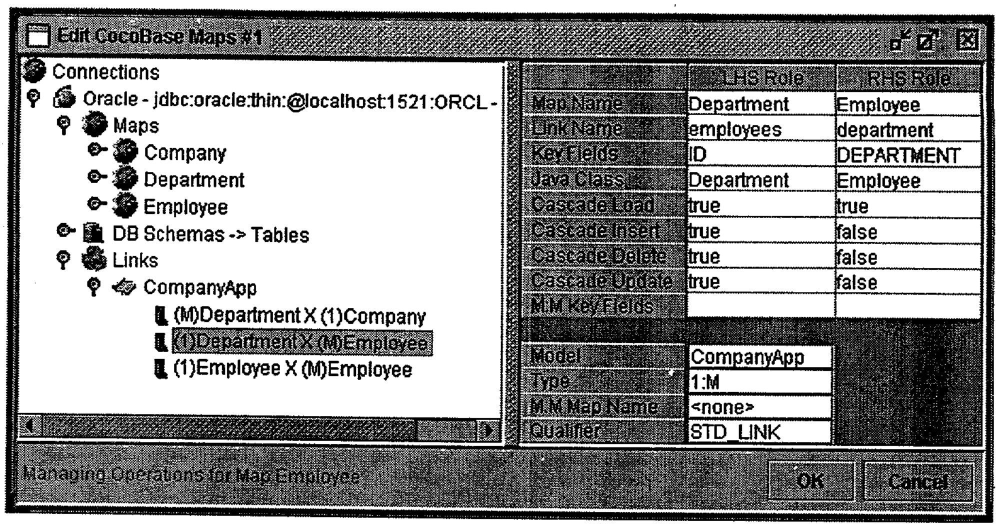

Although totally unconstrained, a Link definition would normally be created for each foreign-primary key relationship in the database tables or, in the case of many-to-many relationships, for each associative table.

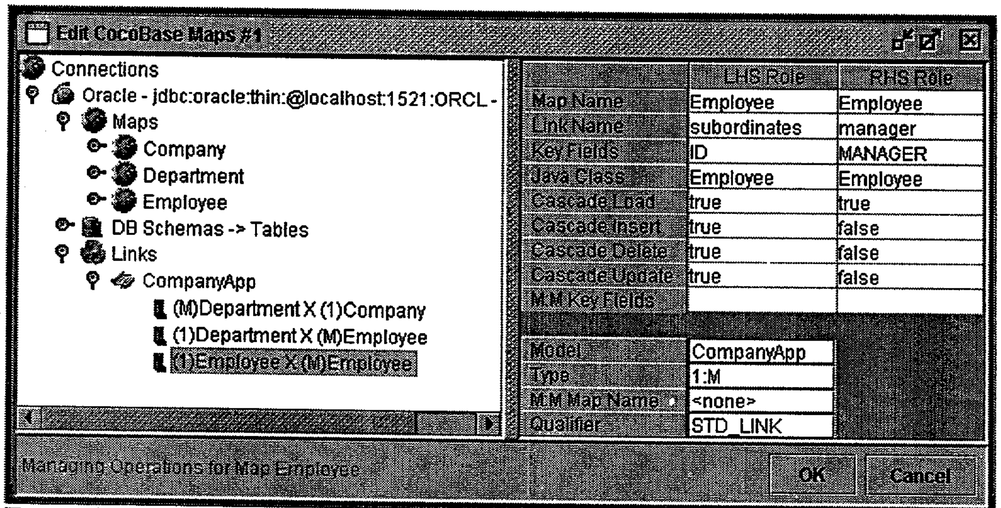
A Link definition is basically a combination of two roles, each having the following information:

**Map Name** – the name of the Map participating in the relationship
**Link Name** – the name of the link that connects to the Map in the opposite role. It normally matches the name of the object property (field or get/set method pair) that holds instances of the related objects.
**Key Fields** – the comma-separated list of fields defined according to the following guidelines:

- If this role is on the "one" side of a one-to-one or one-to-many link, this is the list of the fields in the map that correspond to the primary key defined in the underlying table
- If this role is on the "many" side of a one-to-many link, this is the list of the fields in the map that correspond to the foreign key defined in the underlying table
- If this is a role of a many-to-many links, this is the list of the fields in the map that correspond to the primary key defined in the underlying table

**Java Class** – the full qualified name of the Java class that will be instantiated when the relationship is loaded from the database. This is normally a class corresponding to the Map specified in the opposite role.
**Cascade Load** – indicated if this role is going to load related object instances.
**Cascade Insert** – indicated if this role is going to insert related object instances.
**Cascade Delete** – indicated if this role is going to delete related object instances.
**Cascade Update** – indicated if this role is going to update related object instances.
**M:M Key Fields** – This is relevant only for many-to-many Link definitions and is the comma-separated list of the fields in the associative map that refer to the Key Fields of this role. In this case, the name of associative map must be specified in the box **M:M Map Name**.

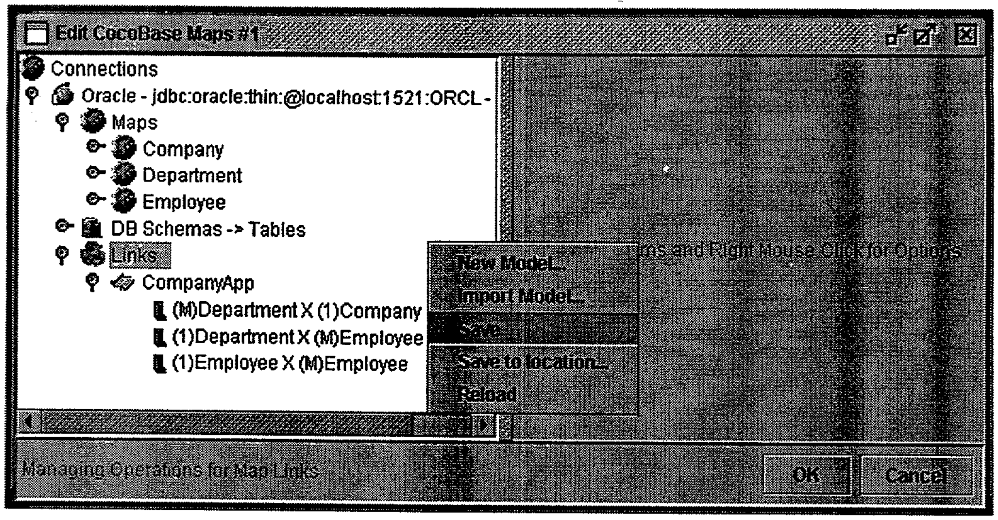



Examples of Link definitions are given above. Click on the appropriate boxes to edit the Link definition for the model being defined. Repeat this procedure for each table relationship that is relevant to the application being developed.

*3) Save Link Definitions*

Once Link definitions are complete, select the branch **Links** in the editor tree right and click on it to access the pop-up menu. Click on the option **Save** to save your work, as show below:




Note that unlike CocoBase Maps, Link models are not kept in the CocoBase repository. Instead, for each Link model, CocoBase will create a property file called *modelname.properties* containing all the Link definitions for that model. This property file is stored under the demos/resources directory (by default - although this can be overridden). As long as the property file describing the Link model is in the classpath (either directly or in a subdirectory called *resources*), CocoBase runtime classed will load the model and register the navigation information.

### CocoBase Transparent Persistence

In the previos sections, it was demonstrated how to map database tables and relationships using CocoBase. In this section, it is shown how CocoBase runtime classes can be used to provide transparent persistence to a Java Object Model which corresponds to the created Maps and Links.

## Transparent Persistence with CocoBase

CocoBase accomplishes transparent persistence with java object models without using bytecode manipulation, proprietary interfaces or class hierarchy intrusion. This means that no special classes or interfaces are needed in the Object Model in order to do persistence with CocoBase. The only requirement is that they must have a default constructor with no arguments.

There are 3 basic Runtime components that are involved in the transparent persistence of objects:

- The CocoBase Runtime O/R mapping class that wrappers the JDBC driver and issues queries and does the actual persistence calls. This is a class such as `thought.CocoBase.CocoPowder` or `thought.CocoBase.CocoPowderPlugin20` (for jdbc 2.0 connections).

- The `thought.CocoBase.Transaction` object that can track changes of instances, and acts as a change 'buffer'. If a Transaction object is used, then it only calls a CocoBase Runtime driver - O/R mapping runtime class - when the txn.commit() is called.

- The `thought.CocoBase.navapi.Navigator` object that can track and detect changes in relationships based on Link definitions.

The `Navigator` class can function in conjunction with a `Transaction` object or it can function standalone. While complex object graphs can be transparently managed directly by the Navigator and without the `Transaction` object, the use of the `Transaction` object is generally preferred because of its buffering and update optimizations which only persist those attributes that have changed.

# Creating Applications with CocoBase Transparent Persistence

Once Links and Maps are properly created, go to **File->Generate Java Code** to generate Java classes. The 'Default CocoNavigate Java Object' code generation target will generate a pure java class with no CocoBase interfaces.

After classes have been generated and compiled, CocoBase runtime classes can be used to persist instances of these classes. First, open a CocoBase connection as follows:

```
CocoDriverInterface myBase = CocoDriver.getCocoDriver(
    "thought.CocoBase.CocoPowder",
    "org.hsql.jdbcDriver",
    "jdbc:HypersonicSQL:hsql://localhost;cocoprop=cocofactory=CocoProxyFactory",
    "sa", ""
)
if(myBase.connect() == -1) {
    System.out.println("Failed connect!");
    System.exit(1);
}
...
```

Then create a CocoBase transaction object to manage any objects that are retrieved. Notice how the transaction object is configured through parameters and property settings.

```
    thought.CocoBase.Transaction cocoTxn =
                              new thought.CocoBase.Transaction(myBase, false);
    Properties cocoTransProps = new Properties();
```

```
cocoTransProps.put("preserveCommit","true");
cocoTransProps.put("commitconnection","true");
cocoTransProps.put("throwExceptions","true");
cocoTransProps.put("updateOnlyChangedColumns","true");
cocoTxn.setProperties(cocoTransProps);

// Begin a new transaction.
cocoTxn.begin();
...
```

Then open a CocoBase Navigator object with the Navigation model to be used and also we register the Transaction object with the Navigation model:

```
    // Instantiate a Navigator with the Link model created from
    // the UML/XMI document
    thought.CocoBase.navapi.Navigator navigator =
                                new thought.CocoBase.navapi.Navigator (myBase,
"company");
    // Assign the current transaction to the Navigator
    navigator.setTransaction(cocoTxn);
```

Now select the top level node to work with. Notice our use of a CocoProxyM class which 'wrappers' the pure java object model class and gives the class the compatibility with CocoBase through java reflection instead of requiring any special interfaces in the java class itself:

```
    // Setup our object to query
    Department dept = new Department();
    department.setName("SALES");

    // This will read & bind all 'Department' objects to the transaction.
    Vector deptVector = myBase.selectAll(

                                new
thought.CocoBase.CocoProxyM(dept),"Department");
```

Now it is possible to step through each of these objects and tell the retrieved object to be navigated through the `loadAllLinks` method which does the automatic navigation for that object:

```
    for(int i=0; i< deptVector.size(); i++) {
                                Department d =
(Department)deptVector.elementAt(i);
                                // Because the cascadeLoad flag is set to true in
the direction
                                // Department->Employees, the employees link will
load automatically
                                d = navigator.loadAllLinks(d,"Department");
                                ...
                                Vector emps = d.getEmployees();
                                for (int j=0; j<emps.size(); j++) {
            // raise salaries by 20%
            emp.setSalary(emp.getSalary()*1.2);
            ...
                                }
                                ...

        // Once changes are made to an object graph those changes can be
        // synchronized using the updateAllLinks method such as:
```

```
                        navigator.updateAllLinks(d,"Department",true);
}
```

You can then commit the buffered changes with:

```
cocoTxn.commit();
```

These code introductions are quite small, and can be done entirely server side with Entity or Session beans in J2EE environments with no model or object model intrusion. And for local non-j2ee apps the application intrusion is incredibly small with only the 'load', 'synchronize' and 'persist' operations requiring a single method call for each root node.

The Navigator supports one-to-one, one-to-many & many-to-many link definitions with cycle detection. It also detects this locally or by reconciling serialized or copied objects without object model or bytecode intrusion. This is truly transparent persistence that's architected and designed for the VM oriented Java language. There are also more advanced applications included in the demos/pguide/navapi subdirectory that demonstrate all of the mapping one-to-one, one-to-many & many-to-many relationships as well as an EJB using the Navigator system to manage a graph of java objects.

**Conclusion**

This document described how to add CocoBase Transparent Persistence to Object Models and applications from SQL database tables. It is only a brief overview of what can be done with CocoBase. The architecture and implementation presented here is uniquely suited to work in every app from the tiny local app to the enterprise J2EE and to do so with superior performance and manageability.

Hope this gets you started and if you have any more questions you can post them to the forum at http://forum.thoughtinc.com or by sending support email to support@thoughtinc.com.